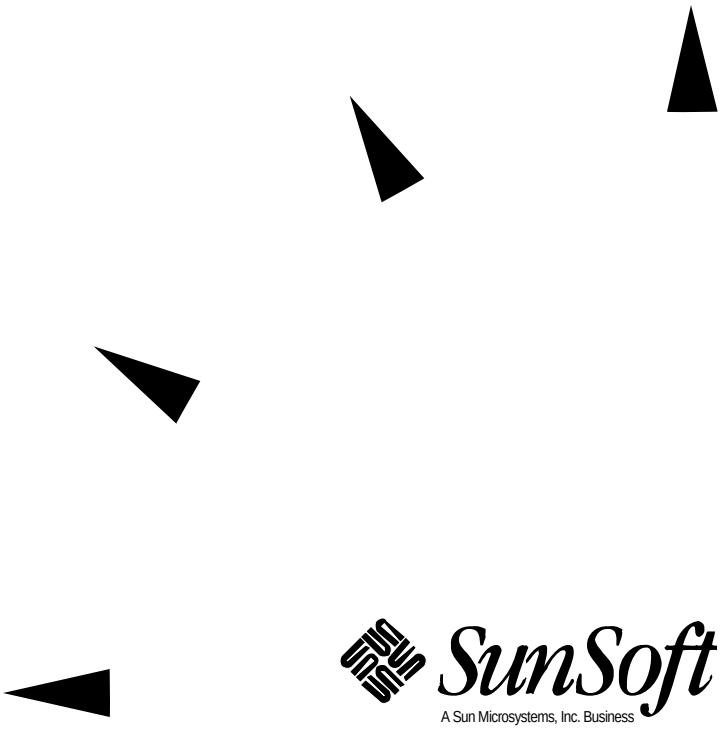
XIL Device Porting and Extensibility Guide



2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Contents

Figures

Tables

Preface

Note – This is an in-progress document that has not had final review. It is subject to change. We appreciate your input on any sections that require correction or further clarification.

This document describes the architecture of, and internal interfaces to, the XIL library. It describes the library's C++ classes and discusses the mechanism for acceleration and porting of new hardware. The functionality of the XIL library is discussed in the documents *XIL Programmer's Guide* and *XIL Reference Manual*.

Who Should Use This Book

This book is designed for people porting hardware to use the XIL imaging library, as well as for people who are writing additional device-independent acceleration code for XIL functions.

Before You Read This Book

It is assumed that the reader is familiar with C++ and the ideas of classes and class inheritance in C++. It is further assumed that the reader has studied the *XIL Programmer's Guide* to become familiar with the capabilities of the XIL library.

What's in This Book?

Chapter 1, "Overview" provides an overview of the XIL library and describes the device-independent classes used to implement the library.

Chapter 2, "More on Writing Device Handlers" provides general information about writing XIL device handlers.

Chapter 3, "I/O Devices" describes how to write I/O device handlers and provides an example implementation of an I/O device handler.

Chapter 4, "Compute Devices" describes how to write compute device handlers and provides an example implementation of a compute device handler.

Chapter 6, "Storage Devices" describes how to write storage device handlers and provides an example implementation of a storage device handler.

Chapter 5, "Compression/Decompression" describes how to add a new compression method and compression hardware, and provides an example implementation of a compressor.

Appendix A, "XilOp Object" lists the number of image sources supported by an XIL function and the `XilOp` member functions that must be used to extract the image sources and to extract an XIL function's parameters from the `XilOp` object.

Appendix B, "XIL Atomic Functions" provides the name of the function that must be supplied in the `XILCONFIG` header comment to associate an implemented function with an API call.

Related Books

XIL Reference Manual

XIL Programmer's Guide

XIL Test Suite User's Guide

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

XIL DDK Directory Structure

The default installation directory for the XIL DDK (Driver Developer Kit) is `/opt/SUNWddk/ddk_2.4/xil`. The structure of the XIL DDK directories is described in Figure P-1 and in the sections that follow.

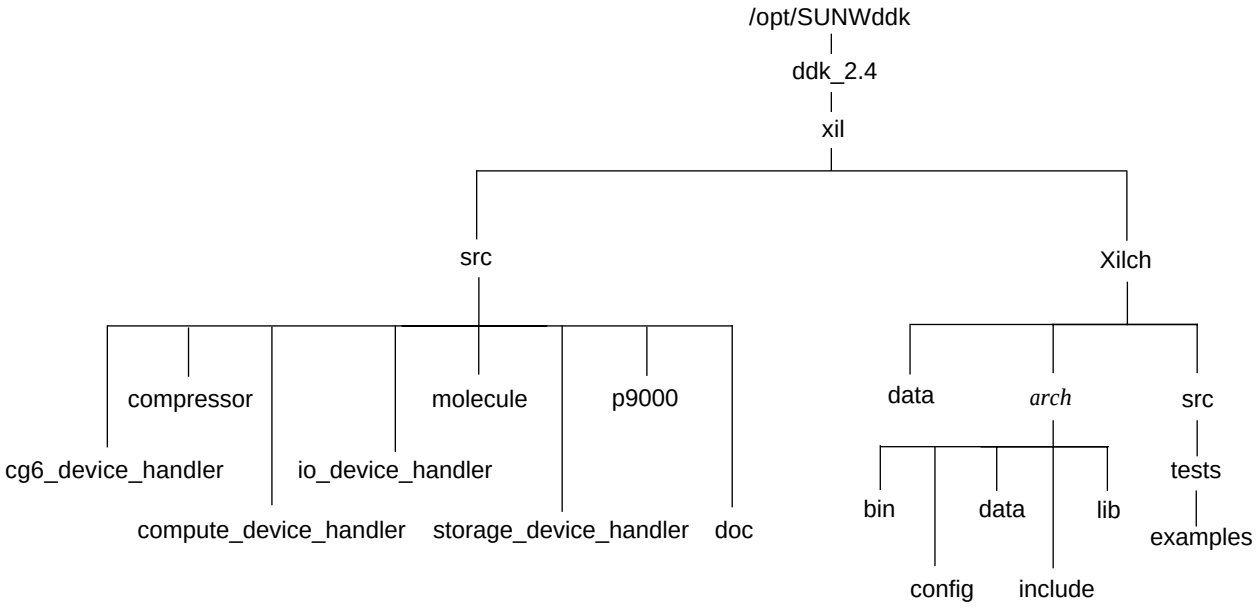


Figure P-1 Directory Structure of XIL DDK Release

src/

The `src` directory contains seven subdirectories of examples; six of these examples are described in this manual: `cg6_device_handler`, `compressor`, `compute_device_handler`, `io_device_handler`, `molecule`, and `storage_device_handler`.

Note – The directory `src/cg6_device_handler` contains an example I/O device handler that treats a SPARC GX frame-buffer window as an I/O device. The directory `src/p9000`, which isn't discussed in this manual, contains an example for an x86-specific module that is based on the SPARC GX example: it treats a p9000 frame-buffer window as an I/O device. The p9000 example isn't discussed in this manual because the p9000 architecture is similar to the CG6 architecture. The p9000 code is included to demonstrate some of the differences you can expect when writing an XIL module for x86.

The `src/doc` directory contains a source (`.po`) file used for generating error messages.

Xilch/

The `Xilch` directory contains the files for the XIL Test Suite, including executables, data files, and examples. The XIL Test Suite is described in the *XIL Test Suite User's Guide*.

Overview 1

This chapter describes the Solaris™ XIL™ Imaging Library. It has the following main sections.

Introduction to the XIL Imaging Library

The Solaris XIL Imaging Library provides a basic set of functions for imaging and video applications. The XIL library is the imaging component of the Solaris Graphics Architecture, a strategy for providing low-level software interfaces known as foundation libraries. Application and API developers can port their code to such foundation libraries. The XIL library is complemented by the OpenGL Graphics Library, which addresses application and API requirements for geometry-based graphics, and the Kodak Color Management System (KCMS™) library, which enables device color management.

1

Solaris Graphics Architecture

The XIL foundation library is an integral part of the Solaris Graphics Architecture. The Solaris software, using loadable drivers, enables display devices using the Solaris Graphics Architecture to be easily installed and used, without requiring kernel modifications. The Solaris Graphics Architecture, through the XIL, OpenGL, KCMS, and X11 software, provides a means for third-party hardware and software vendors to develop applications with the knowledge that their investment will see long-term benefits, including access to a range of computing platforms and complete integration into the Solaris environment.

Note – Currently, the compression GPI interfaces to the XIL library discussed in this book are not binary committed. Due to the evolving nature of the C++ language, these interfaces may change in ways that may require you to change your code and recompile in a later Solaris release. However, the compute, I/O, and storage GPI interfaces are now committed, which means that we will maintain source-code compatibility. The API continues to be committed for binary compatibility.

Division of Function in the XIL Library

The XIL architecture consists of a high-level application programming interface (API), device-independent *core* code (including the XIL API and GPI layers), which manages the loading and calling of specific device-dependent functions, a graphic porting interface (GPI), which separates device-independent and device-dependent code, and the device-dependent (DD) algorithm implementation. Figure 1-1 illustrates this division of function and shows how these sections relate:

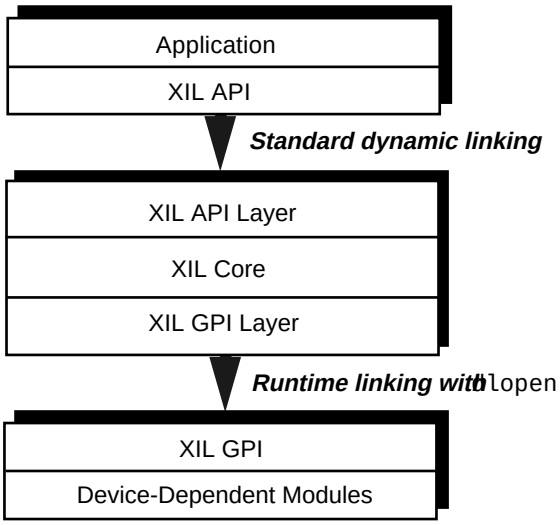


Figure 1-1 The XIL Internal Architecture

This document describes the XIL core (including the XIL API and GPI layers), the GPI, and the method needed to supply alternative DD code. In general, porting new hardware to the XIL environment involves providing new implementations of DD modules. The GPI is the interface through which the DD modules are called and is responsible for allowing the creation of new DD implementations without requiring exposure of XIL library source code.

XIL API Layer

The API layer in the XIL library contains the C wrappers on the C++ device-independent classes. It consists of functions that can be categorized in the following way:

- Create and destroy objects
- Set and get object attributes
- Modify image data
- Extract information from an image
- Modify data in non-image objects
- Synchronize operations

The semantics of the functions exposed in the API are described in the *XIL Programmer's Guide* and the *XIL Reference Manual*.

The C++ device-independent classes are used to implement the API functions described above. These classes provide a device-independent interface to the XIL library imaging functionality and are primarily used to pass information through the GPI to the DD modules. Table 1-1 briefly summarizes these classes and presents them alphabetically as base classes or API-level classes. For additional information on each class, see "XIL Base Classes" on page 23, and "XIL API Level Classes" on page 24.

Table 1-1 XIL C++ Device-Independent Classes

Class Name	Description
XIL Base Classes	
`XilGlobalState`	Contains a list of system states and the tree of atomic/molecular operations and their corresponding function pointers
`XilObject`	Is the parent class for all XIL API classes
`XilSystemState`	Contains the creation methods for all API classes
XIL API Level Classes	
`XilCis`	Contains the compressed image data and compression functions
`XilColorspace`	Contains information to specify a color space
`XilColorspaceList`	Contains information for specifying a list of color spaces
`XilDevice`	Contains multiple device attributes
`XilDitherMask`	Contains dither matrices for ordered dithering
`XilError`	Contains information for reporting errors
`XilHistogram`	Contains image histogram information
`XilImage`	Contains the basic data elements for XIL functions
`XilImageFormat`	Contains an image description without data
`XilInterpolationTable`	Contains an array of 1 x *n* kernels that represent the interpolation filter in either the horizontal or vertical direction

Table 1-1 XIL C++ Device-Independent Classes *(Continued)*

Class Name	Description
`XilKernel`	Contains kernel data used in convolution, error diffusion, paint, and band combine
`XilLookup`	Contains data for image conversion and colormap use
`XilRoi`	Contains region of interest information for an image
`XilSel`	Contains a structuring element used in erosion and dilation
`XilStorage`	Contains storage format information for an image

XIL Base Classes

The `XilGlobalState` *Class*

One instance of `XilGlobalState` exists for each heavyweight XIL process.

This class is responsible for loading DD code as needed by the application. This class contains a table that converts operation numbers to operation names and vice versa.

The `XilGlobalState` class is defined in the `_XilGlobalState.hh` header file.

The `XilObject` *Class*

`XilObject` is the base class from which all API level XIL objects are derived. It is the parent of the `XilDeferrableObject` and `XilNonDeferrableObject` classes from which the API level XIL objects are directly derived.

The `XilObject` class contains all the attributes and functions that are generic to the exposed objects. It is an abstract class; no instance of this class is ever created.

Each API level object has a 64-bit object number and version number. The combination of these two numbers indicates a unique version of the same object and can be returned for any `XilObject` derived class using the member function `getVersion()`. A copy of an object returns the same version number.

When API-level objects are modified, their version number is changed. Use of object versions allows intelligent caching of API objects within the implementation.

This `XilObject` class is defined in the `_XilObject.hh` header file.

The `XilDeferrableObject` *Class*

The `XilDeferrableObject` class is used to derive those objects that may be deferred, such as the `XilImage` and the `XilCis`. These objects have common characteristics of the deferrable execution information. For more information on deferral of objects, see "Deferred Execution" on page 30.

The `XilNonDeferrableObject` *Class*

The `XilNonDeferrableObject` class contains all those objects that are not deferrable such as the `XilRoi`, `XilLookup`, `XilHistogram`, and so forth.

The `XilSystemState` *Class*

The `XilSystemState` class contains the common information for an individual XIL session initiated via `xil_open()`. This class can be used to create objects. All objects have a reference to the system state that created them. The system state is also used to generate errors via macros defined in its header file.

The `XilSystemState` class is defined in the `_XilSystemState.hh` header file.

XIL API Level Classes

The sections below describe the API level classes. As stated in "The XilObject Class" on page 23, `XilObject` is the base class for all the API level classes. You cannot instantiate an object from these classes; instead, you must get a copy of or a reference (pointer) to the object. If you get a copy, you are responsible for freeing the allocated data.

The `XilCis` *Class*

The `XilCis` (for compressed image sequence) class is the primary object for compression in the XIL library. It contains member functions to allow access to and movement through compressed data. The `XilCis` is created by loading a specified compressor.

The `XilCis` class is defined in the `_XilCis.hh` header file.

The `XilColorspace` *Class*

`XilColorspace` describes a color space of an image in such a way that images may be transformed from one color space to another. The XIL Imaging Library supports ten named color spaces. `XilColorspace` may contain a KCMS profile which is used by `xil_color_correct()`, not by `xil_color_convert()`.

Each of the named color spaces is identified as an `XilColorspaceOpCode`. `XilColorspaceOpCode` is an enumeration type with the values shown below:

- `XIL_CS_RGBLINEAR`
- `XIL_CS_RGB709`
- `XIL_CS_PHOTOYCC`
- `XIL_CS_YCC601`
- `XIL_CS_YCC709`
- `XIL_CS_YLINEAR`
- `XIL_CS_Y601`
- `XIL_CS_Y709`
- `XIL_CS_CMY`
- `XIL_CS_CMYK`

The `XilColorspace` class is defined in the `_XilColorspace.hh` header file.

The `XilColorspaceList` *Class*

The `XilColorspaceList` class contains information for specifying a list of color spaces.

The `XilColorspaceList` class is defined in the `_XilColorspaceList.hh` header file.

The `XilDevice` *Class*

The `XilDevice` class describes the attribute/value pairs of a device. The member functions of this class enable you to access and set a device's attributes.

An `XilDevice` object can be used to set multiple device attributes simultaneously. This is important when device images are created and when the setting of device attributes incurs substantial overhead.

When you use this object to create a device, you should set only attributes the device understands. If the device does not recognize an attribute that you have set through the `XilDevice` object, an error is generated. You should set default values for a device's attributes based on the list of attribute/value pairs returned by the `XilDevice` object.

The `XilDevice` class is defined in the `_XilDevice.hh` header file.

The `XilDitherMask` *Class*

In the simplest case, the dither mask is a two-dimensional array of values that determines how the noise added during the dither process is spread across the image. In the XIL library, the dither mask can have multiple bands, each band with its own matrix. This allows noise to be spread differently for each channel of a true-color image, which can enhance the result of the dither operation. For dithering of multiband images, the number of bands in the dither mask matches the number of bands in the source image.

The `XilDitherMask` class is defined in the `_XilDitherMask.hh` header file.

The `XilError` *Class*

The XilError class describes errors in the XIL library. Its member functions allow programs to get information about the error, to retrieve the object that is associated with the error, and to control the error handling routines.

The `XilError` class is defined in the `_XilError.hh` header file.

The `XilHistogram` *Class*

The `XilHistogram` class describes a multidimensional histogram. This object can be used to gather statistical information on images.

The `XilHistogram` class is defined in the `_XilHistogram.hh` header file.

The `XilImage` *Class*

Derived from the `XilImageFormat` class, `XilImage` represents an image along with its associated data. The `XilImage` class contains member functions that make up the XIL image functions. It also contains member functions for storage and retrieval of image attributes.

The `XilImage` class is defined in the `_XilImage.hh` header file.

The `XilImageFormat` *Class*

The `XilImageFormat` class carries information about an image, independent of its associated pixel values.

Note – This class is equivalent to the API class `XilImageType`.

`XilImageFormat` is used at the API level to return information about the kind of image the application should create to act as a destination from a decompression or device capture, or as a source to a compression or device display. It is subclassed by the XIL library to create the `XilImage` class. There is an API function that creates an image directly from an `XilImageFormat` object.

The `XilImageFormat` class is defined in the `_XilImageFormat.hh` header file.

The `XilInterpolationTable` *Class*

The `XilInterpolationTable` class supports general interpolation. See the *XIL Programmer's Guide* for a discussion about general interpolation. This class describes an array of 1 x *n* kernels. The array represents the interpolation filter in either the horizontal or vertical direction. The member functions of this class enable you to access the data in an `XilInterpolationTable` object.

The `XilInterpolationTable` class is defined in the `_XilInterpolationTable.hh` header file.

The `XilKernel` *Class*

The `XilKernel` class represents a two-dimensional array of floating point values. `XilKernel` objects are used as parameters in functions like image convolution and error diffusion.

The `XilKernel` class is defined in the `_XilKernel.hh` header file.

The `XilLookup` *Class*

The `XilLookup` class describes data that is used to interpret image data. `XilLookup` is defined in the `_XilLookup.hh` header file.

Three classes are derived from `XilLookup`. Table 1-2 lists these classes and the header files in which they are defined.

Table 1-2 `XilLookup` Subclasses

Class Name	Header File
`XilLookupColorCube`	`_XilLookupColorcube.hh`
`XilLookupCombined`	`_XilLookupCombined.hh`
`XilLookupSingle`	`_XilLookupSingle.hh`

The `XilRoi` *Class*

`XilRoi` describes an arbitrary region of interest (ROI). Member functions exist to manipulate and logically combine XIL ROIs.

A ROI has three internal representations:

- A list of rectangles (set and get as rectangles by the user)
- A bitmask (set as an image by the user)
- A list of convex regions (accessible by the core and GPI only)

Translation between types is done as needed and stored in the ROI until invalidated by subsequent changes to the ROI.

The `XilRoi` class is defined in the `_XilRoi.hh` header file.

The `XilSel` *Class*

The `XilSel` class describes a structuring element, which is a two-dimensional description of a pixel neighborhood. In the XIL library, the structuring element is described with a two-dimensional boolean (integer) array, with pixels in a neighborhood having true values in the array, and pixels excluded from the neighborhood having false values. Structuring elements are currently used as parameters to the `xil_erode()` and `xil_dilate()` functions.

The `XilSel` class is defined in the `_XilSel.hh` header file.

The `XilStorage` *Class*

The `XilStorage` class describes a contiguous region of memory associated with a given image. At construction, an `XilStorage` object contains no information but is filled in by subsequent calls. For improved performance and to ensure that an `XilStorage` object is destroyed automatically when no longer needed, an `XilStorage` object should be constructed on the stack.You fill in `XilStorage` objects by calling the `XilDeferrableObject::getStorage()` and `XilImage::getStorage()` methods for the appropriate images.

XIL Core Layer

The Core layer in the XIL library manages the dynamic loading of device handlers, deferred execution, and operation scheduling.

Deferred Execution

The primary problem in achieving adequate performance in an imaging library comes from the way in which the units (or atoms) of functionality are arbitrarily combined to perform useful work. The typical imaging case is much more general than, for example, OpenGL with its well-defined processing pipeline. This has tended to limit the usefulness of general imaging libraries, since any reasonable division of imaging functionality into atoms renders the performance of many applications substandard. The result is that useful libraries tend to be closely tailored to applications and vertical markets.

The use of multiple passes of atoms impedes performance in at least three ways:

- Multiple passes through an image cause the entire image to be paged into memory multiple times. Since in many cases the images are large compared with available physical memory, and the application is often working with multiple images simultaneously, this significantly impairs performance. Often a pixel operation can be performed in a single CPU clock cycle, so the time spent getting to the data far outweighs the time needed for the operation. Imaging is often a worst case of I/O bound processing.
- Many combinations of atoms can be performed in a single logical step with little penalty. For example, in the case of a rotation followed by a zoom, the backward-mapped algorithms often used to perform the rotation can perform both operations in nearly the same time as the rotation alone.
- The application must often create temporary images to hold intermediate results. Such intermediate images are not needed in customized code and may be avoided if the operations can be combined.

The XIL Library Method

There are several methods that can address these problems. In the XIL library, we have chosen to implement deferred execution and multiple atomic operation replacement. The goal is to identify and replace a sequence of atomic operations with a functionally equivalent single operation (a *molecule*). In this approach, the core-layer code keeps track of image dependencies and causes the operations to occur as late as possible. This enables significant performance improvements as described below.

In the library, atomic functions are, by default, deferred as long as possible. To implement this, the API level function creates an instance of the `XilOp` class (described in "The XilOp Class" on page 39), adds the API parameters to the `XilOp`, and then places the operation on a tree-like structure that holds deferred operation information. `void` is then returned to the calling routine (the C binding in this case).

The deferred execution data is stored as a *directed acyclic graph* (DAG), where the nodes are the instances of the `XilOp` class described above. The fact that a destination function depends on its sources is stored, along with the operation and parameters necessary to produce the destination image once the sources are produced. As image results are needed, the parts of the graph that hold that information are evaluated. Their dependent images are generated by performing the operations that have been stored.

Several actions can cause the evaluation of a subgraph:

- A request for results by the application—either from an I/O image or through storage acquisition
- A call to `xil_state_set_synchronize()`which modifies all images in the library to run synchronously
- A call to `xil_set_synchronize()`that turns on synchronization for an image

The DAG is disassembled upon a call to `xil_close()`.

Graph Evaluation and Molecules

When the graph is evaluated, each node's op (the operation used to produce the node's destination image) is available and could be used to index into a list of function pointers. In fact, the library does something a little more general than this, and thus gains the ability to accelerate combined operations.

The XIL library stores its function table as an array of trees, each tree having one of the atomic functions as its base. Branches exist from the base node describing each composite operation (*molecule*) that exists. This structure is built from the description of the contents of each compute device handler. As the core code looks at the DAG, it attempts to match the longest sequence of atoms in the DAG to the function table. If the needed molecule is available, it is called; otherwise, the sequence of functions is checked again, leaving off the last function, which is performed atomically.

Each node on the function tree is a list of possible functions, usually using different compute devices. The core code calls the highest priority function, which is assumed to be the optimal (accelerated) one. The accelerated function is allowed to fail gracefully, in which case the second function in the list is called. Typically, the last function in the list is the unaccelerated *memory* port, which is guaranteed to work for all cases. This construction allows an IHV to accelerate a function for only a subset of the input parameters. For example, the code supporting an accelerator that only scales images up can fail gracefully (and cause the memory function to be called) if the scale factors it pulls off the DAG are less than unity. The mechanism for inserting a function into the table is described later in this document (see "Registering Operations With the XIL Library" on page 73).

The core code does not require porting.

When porting devices, you can accelerate either atomic functions or molecules. You can create molecules from any combination of atomic functions; however, you cannot add new atomic functionality to the library.

Some Considerations

The time needed to determine the sequence of operations from the DAG and choose the appropriate function from the table is trivial compared to typical image operations.

Not all operations can be deferred. An example of this is the `xil_extrema()` function, which supplies the maximum and minimum image values. The library makes no effort to hide the values returned in an opaque structure, the contents of which could be deferred. Thus, the use of `xil_extrema()` causes an evaluation of the source image. In general, only the functions that have as their destination an `XilDeferrable` object such as `XilImage` or `XilCis` object (or create those objects) can be deferred. The complete list of the rules for deferred execution is as follows:

1. Functions that return information based on values in the current image cannot be deferred. These functions are:

- `xil_choose_colormap()`
- `xil_extrema()`
- `xil_histogram()`
- `xil_lookup_convert()`
- `xil_squeeze_range()`

2. Functions that have nonstandard ROI, origin, or size behavior cannot be deferred. These functions are listed below:

Note – The `xil_scale()` and `xil_transpose()` functions may be deferred under special circumstances. See the *XIL Programmer's Guide.*

- `xil_affine()`
- `xil_paint()`
- `xil_rotate()`
- `xil_scale()`
- `xil_subsample_adaptive()`
- `xil_subsample_binary_to_gray()`
- `xil_tablewarp()`
- `xil_tablewarp_horizontal()`
- `xil_tablewarp_vertical()`
- `xil_translate()`
- `xil_transpose()`

3. General rules that apply to the other XIL functions are as follows:

Note – The `xil_copy_pattern()` function is exempt from general rules b and c below.

---

   a. The source and destination images must have the same ROI.

   b. The source and destination images must have the same origins.

   c. The source and destination images must have the same width (xsize) and height (ysize).

   d. The source images cannot have the same parent as the destination image.

We do not envision a large number of molecules in a typical release. In particular, `display(zoom(decompress()))` molecules have proven to be advantageous. Other display pipelines (`display(zoom())`, `display(dither(zoom()))`, etc.) will prove useful. It is expected, however, for a third party to add molecules that particularly benefit its vertical market, without requiring that other software running on the XIL library be modified.

One goal of deferred execution is that the application need never know when functions are actually performed. Asynchronous error reporting allows this to be the case in general. However, some cases are impossible to hide. Consider

the case of a frame grabber used as a source to an operation that is done in response to an external signal. In normal operation, the actual grab would be postponed until the dependency tree was evaluated, possibly several steps further in the program. A possible resolution to this is to make the destination of the grab operation synchronized. This causes the grab to occur when the function call is made, but precludes any optimization of the grab function. In the end, the application must choose whether the operation should be deferred or not, and when the synchronization should occur. With the general rule “no optimization through synchronization,” the application writer can judge an appropriate place to synchronize.

Molecules must behave semantically like the sequence of atomic operations, and produce the same (or nearly the same) results as calling the individual atomic functions. A molecule cannot have a greater precision than the atomic functions that the molecule contains. For example, a molecule of two `XIL_BYTE` convolutions cannot use floating point between the convolutions. It must clamp the intermediate results to `XIL_BYTE` images. An alternative molecule would be:

```
cast;8->f,convolve;f,convolve;f,cast,f->8
```

See Appendix B, “XIL Atomic Functions, “ for information on the syntax of atomic functions.

Unusual Effects of Deferred Execution

One effect of deferred execution is that in some cases source code may not accurately reflect the actual operations done. Consider the following case, where `im2` is not set to be synchronous.

```
for (i=0; i<N; i++) {
    a[0] = i;
    xil_add_const(im1, a, im2);
}
xil_copy(im2, display_image);
```

In the XIL library, only one add (the last one) is done as a result of this code, since the earlier results are obscured by the later ones. If the final copy were not called, no evaluation of the add would take place at all. In normal code, such cases rarely arise, but one must be careful in benchmarking the library. This is not unlike the situation that occurs with optimizing compilers.

Consider another case where only the final decompress is executed.

```
while (xil_cis_has_frame(cis)) {
    xil_decompress(cis,im2);
}

xil_copy(im2,display_image);
```

Each call to `xil_decompress()` schedules a frame from `cis` to be decompressed into image `im2`. This destination image is not used until the decompress loop is exited. The last decompressed frame is copied to a display image; this is the only operation that is evaluated.

XIL GPI Layer

The GPI layer is the interface for device-dependent code. In general, porting a device to the XIL library requires subclassing one or more of the base device classes defined below, and then configuring the resulting object files so that they can be loaded at run-time by the library. In addition to enabling third parties to port hardware, the functions and device access in the standard XIL release are provided through this interface as well.

Table 1-3 lists the classes that are defined in the GPI layer.

Table 1-3 XIL GPI Layer Classes

Class Name	Definition
`GPI LAYER DEVICE CLASSES`	
`XilDeviceManager`	Is the base class for the device type object
`XilDeviceManagerIO`	Is the abstract class for I/O devices
`XilDeviceIO`	Is the device-specific base class for I/O
`XilDeviceManagerCompute`	Is the abstract class for compute devices
`XilDeviceManagerCompression`	Is the abstract class for compression devices
`XilDeviceCompression`	Is the device-specific class for compression devices
`XilDeviceManagerStorage`	Is the abstract class for storage devices

Table 1-3 XIL GPI Layer Classes *(Continued)*

Class Name	Definition
`XilDeviceStorage`	Is the device-specific base class for storage
`GPI LAYER SUPPORTING CLASSES`	
`XilBox`	Represents the area to be processed
`XilBoxList`	Represents a list of destination areas and their corresponding source area
`XilCondVar`	Contains wrappers for conditional variables
`XilConvexRegionList`	Contains information for representing a list of convex regions.
`XilFunctionInfo`	Holds the information for a device to describe a function to the XIL library
`XilMutex`	Contains wrappers for mutex locks
`XilOp`	Holds the information required to store the operation on the DAG
`XilRectList`	Contains information for generating a rectangular list
`XilScanlineList`	Contains information representing a scanline list
`XilTile`	Contains the information representing a tile
`XilTileList`	Contains a list of tiles

GPI Layer Device Classes

The XIL library has the concept of devices—software and hardware—which are represented by the device handler modules. More than one instance of a device may be created. In this case, information that is common to all instances of a device should be held in the `XilDeviceManager` class.

The `XilDeviceManager` *Class*

`XilDeviceManager` is an abstract class, containing the general information needed at this level.

The XilDeviceManager class is defined in the _XilDeviceManager.hh header file.

XilDeviceManager *SubClasses*

The XIL library subclasses XilDeviceManager for each kind of device that is supported. Devices combine with the appropriate XilDeviceManager subclasses to implement all pipelines. Table 1-4 lists the subclass for each device type and the header file in which each is defined.

Table 1-4 XilDeviceManager Subclasses

Subclass	Header File
XilDeviceManagerIO	_XilDeviceManagerIO.hh
XilDeviceManagerCompute	_XilDeviceManagerCompute.hh
XilDeviceManagerCompression	_XilDeviceManagerCompression.hh
XilDeviceManagerStorage	_XilDeviceManagerStorage.hh

Device-Specific Base Classes

Table 1-5 lists the device-specific base class for each device type and the header file in which each is defined.

Table 1-5 Device-Specific Base Classes

Device-Specific Class	Header File
XilDeviceIO	_XilDeviceIO.hh
XilDeviceCompression	_XilDeviceCompression.hh
XilDeviceStorage	_XilDeviceStorage.hh

GPI Layer Supporting Classes

The XilBox *Class*

The XilBox class holds all the information needed to represent the area of an image that is to be processed.

The class contains member functions to retrieve the coordinates of an image area from the box, however most routines will simply pass an XIL box on as arguments to other functions.

The `XilBox` class is defined in the `_XilBox.hh` header file.

The `XilBoxList` *Class*

The `XilBoxList` class holds all the information needed to represent a destination area and its corresponding source areas for processing.

The class contains member functions to retrieve all the boxes from the list, one set at a time, until all boxes are processed.

The `XilBoxList` class is defined in the `_XilBoxList.hh` header file.

The `XilCondVar` *Class*

The `XilCondVar` class provides support for thread control within a compute routine.

The `XilCondVar` class is defined in the `_XilCondVar.hh` header file.

The `XilConvexRegionList` *Class*

The `XilConvexRegionList` class contains information for representing a list of convex regions when backward mapping from a destination image to a source image generates non-rectangular regions (such as in an affine operation). For more information on convex regions, see the subsection entitled “Affine” on page 112 in “Geometric Operations.”

The `XilConvexRegionList` class is defined in the `_XilConvexRetion.hh` header file.

The `XilFunctionInfo` *Class*

The `XilFunctionInfo` class contains member functions that store the descriptions of device function capabilities. It is set when adding a function via `addFunction()`.

The XilFunctionInfo class is defined in the _XilFunctionInfo.hh header file.

Note – It is recommended that device providers use the xilcompdest.pl script, which takes care of describing functions to the XIL library. For more information, see "Registering Operations With the XIL Library" on page 73.

The XilMutex *Class*

The XilMutex class provides support for thread locking within a compute routine.

The XilMutex class is defined in the_XilMutex.hh header file.

The XilOp *Class*

The XilOp class contains all the information representing a specific XIL operation. The class contains all the information to define a particular imaging function including but not limited to:

- Operations such as ROI manipulation
- Backward mapping from a destination point to a point in the source
- Whether an operation can be forward mapped
- Forward mapping from a point in the source to a point in the destination
- A pointer to a list of other operations if this operation is part of a sequence

The XilOp class describes an operation completely. Any deviation from what is described for an operation in an XilOp class is, by definition, a distinct op.

Atomic operations may have up to three source images and one destination image. However, the interface is generalized to support more diverse atomic operations. You can extract the source and destination images or CISs as well as the other parameters of an XIL operation—all of which are stored in an XilOp object—by using XilOp member functions. See Appendix A, "XilOp Object," for a list of all the source and destination images and parameters for each XIL operation and an description of how to use XilOp member functions to extract this information from an XilOp object.

A molecule is a chain of atomic operations. For a molecule, you must follow the chain properly to extract in a logical order the parameters and images from the XilOp object. The op passed to the routine is the op associated with the

last operation in the chain (the operation that writes its output to a destination image). It contains a pointer to an ordered list of `XilOp` objects, which allow you to pick up parameters for previous ops in the chain.

For more information on molecules and operation chains, see “Operation Prototype: Molecule” on page 62.

The `XilOp` class is defined in the `_XilOp.hh` header file.

The `XilRectList` *Class*

The `XilRectList` class contains information for generating a rectangular list. For more information on using `XilRectList` objects, see “Geometric Operations” on page 109.

The `XilRectList` class is defined in the `_XilRectList.hh` header file.

The `XilScanlineList` *Class*

The `XilScanlineList` class contains information representing a scanline list. An `XilScanlineList` object provides a convenient way for geometric operations to turn a convex region into a list of scanlines. For more information on using `XilScanlineList`, see “Geometric Operations” on page 109.

The `XilScanlineList` object is defined in the `_XilScanlineList.hh` header file.

The `XilTile` *Class*

The `XilTile` class contains information representing a tile.

The `XilTile` class is defined in the `_XilTile.hh` header file.

The `XilTileList` *Class*

The `XilTileList` class contains a list of tiles.

The `XilTileList` class is defined in the `_XilTileList.hh` header file.

Writing Device Handlers

Chapter 2, “More on Writing Device Handlers,” discusses information that generally applies to all devices. Chapters 3 through 6 discuss information specific to each device type.

I/O Devices

I/O devices include any devices that can produce or display images, such as scanners, frame grabbers, image files, and displays. Configured I/O devices appear as “device images” to XIL applications, and may be used as sources and destinations for all XIL imaging operations. These devices are described in Chapter 3, “I/O Devices.”

Compute Devices

Compute handlers contain the device-dependent implementation of one or more atoms or molecules. For example, a compute device might implement the geometric operators accelerated by an add-on card, or might provide a combination of frequently used functions in the form of a molecule. A compute device may be hardware specific, or may be a software-only implementation of a superior algorithm. Compute handlers are loaded during the first call to `xil_open()`. These handlers are described in Chapter 4, “Compute Devices.”

Compression Devices

Compression devices contain most of the utility functions for implementing a method of compression and decompression, even though the actual compress and decompress functions are provided in an associated compute device handler. The compression device performs buffer management and implements the semantics of the `XilCis` object. A compression device for a specified compressor is loaded when `xil_cis_create()` is called. Compression devices are discussed in Chapter 5, “Compression/Decompression.”

Storage Devices

Storage devices allow images to reside in other places besides host CPU memory. Such a device is typically associated with a compute device, allowing an accelerator to take advantage of image data remaining local to the accelerator during sequential function calls.

The handlers for storage devices are responsible for allocating, deallocating, and describing the data format of the storage on their devices. They are also responsible for data conversion between storage devices. In addition, it is useful to have the storage handler perform single-pixel access for `xil_get_pixel()` and `xil_set_pixel()` to avoid having to convert image data in those cases.

Typically, a compute device handler causes the storage device handler for the device to be loaded when it first tries to create an image on the device. The CPU memory storage handler is loaded at the time of the first image creation. Storage devices are discussed in detail in Chapter 6, “Storage Devices.”

More on Writing Device Handlers 2

This chapter provides basic information on writing device handlers that use the XIL library.

What Does the XIL Library Provide?

The XIL library provides software implementations of all atomic functions and default implementations of some molecules. In addition, it supports four device handler types. Through creation of one or more device handlers, you can provide alternate implementations for any of the XIL-provided atomic functions or molecules, or any additional molecules that you may define.

An atomic function is a basic XIL function. A molecule is composed of more than one atomic function.

You can port functions to provide hardware acceleration or access to your particular device. Porting is discussed in the sections that follow.

What Kinds of Ports Are Possible in the XIL Library?

The mechanism for porting in the XIL library allows you to decide which functions would provide the maximum benefit for your customers. If an add-on card is only good at geometric operators, only those functions need to be ported; the memory versions of the remaining functions are called automatically. If the device is a general-purpose imaging accelerator, you may find it reasonable to provide a compute handler for most or all of the possible XIL atomic functions.

If only a compute handler is written, the XIL library expects that an image ends up residing in the CPU memory after each operation. If an accelerator has its own memory, it is often an advantage to allow the image data to reside on the device between operations. This avoids the overhead of having to copy the data back to the CPU after each operation. The XIL library has the concept of a storage handler, which is a set of functions which implements a copy to and from the specific device. If a storage handler is written, the XIL core code allows the image to reside in accelerator memory until another function requests that it be moved somewhere else. Writing a storage handler can greatly speed up a port for certain types of accelerator devices.

Additional molecules may be implemented by combining atomic functions in ways that accelerate specific application areas. Faster implementations of atomic functions can be used in place of the default implementation. While not properly a *device port*, molecules can greatly improve the performance of groups of operations.

For devices that act as either a source or destination image in an operation, the XIL library has the concept of an I/O handler. Once the handler is written, the application programmer can use the I/O device as a source or destination through the device image mechanism. The I/O device handler may also provide image processing for the source or destination image.

A single device may be represented by more than one handler. For example, an input frame grabber that has integrated processing support can be described by an I/O handler and an associated compute handler. If it appears as though multiple processing operations will be done often on the grabbed images, a storage handler can be written for the frame-grabber board as well.

Compression devices must implement the compression but may be associated with other compute, storage, or I/O handlers as well.

A chapter in this guide describes the details of each type of handler.

What Kinds of Ports Are Not Possible in the XIL Library?

The major constraint on porting in the XIL library is that the set of atomic functions may not be extended by the user. All molecules, including those going to I/O hardware, must be made up of groups of the atomic functions that the XIL library defines and implements. The list of available atomic functions is given in Appendix B, "XIL Atomic Functions."

In addition, the IHV should not change the meaning of existing atomic functions; a new implementation should do exactly what the original version does. The correctness of a new function can be tested using the XIL Test Suite.

Porting of functions not defined by the XIL library must be performed using the mechanism defined by `xil_export()`.

The Development Environment

The porting interface for the XIL library is written in C++. Because C++ compilers lack a stable binary interface, it is important that you write device handler code with the same compiler as the interface part of the library. Two compilers are supported: SPARCompiler™ C++ *4.2* and ProCompiler™ C++ *4.2*.

A compiler flag selects the C++ Application Binary Interface (ABI) used by the XIL library. The flag is added to the compile line and is:

```
-Qoption ccfe -abi=1:4.2:1
```

The XIL library contains the XIL Test Suite. It enables you to perform regression tests against proven reference signatures. The XIL Test suite is described in *XIL Test Suite User's Guide*, which is part of this software release.

The environment variable XIL_DEBUG can be useful in development situations. The options for XIL_DEBUG are described in Table 2-1.

Table 2-1 XIL_DEBUG Options

XIL_DEBUG **Option**	**Definition**
link*xx*	Add the two characters following the option link to the base name of the loadable handlers. This option is especially useful when you want to load a debug version of a handler. For example, link_g causes xilioxlib_g.so.1 to be loaded. If this version does not exist, the handler with the standard name (xilioxlib.so.1) is loaded.
show_action	Print XIL_ACTION, the name of the device (such as XilDeviceComputeMemory), and the name of the function being called to execute each XIL operation (such as setvalue8()), for example: XIL_ACTION[XilDeviceComputeMemory]:setvalue8()
set_synchronize	Disable deferred execution.
provide_warnings	Have the default error handler also output warnings.
use_stripping	Make stripping the tiling mode.
txsize=*	Set the default X tile size (0 = default tiling).
tysize=*	Set the default Y tile size (0 = default tiling).
threads=*	Override the number of threads to create for this machine. This value normally is dependent on the number of processors in the machine. Setting the value to 1 turns off threading.
split_threashold=*	Override the minimum Y tile size for thread splitting.

Multiple variables may be set at once. For example, you could set XIL_DEBUG to show_action:set_synchronize.

Installing XIL Device Handlers

XIL picks up the software pipelines provided by Sun from

`/usr/openwin/lib/xil/devhandlers`

Any machine specific libraries, such as those provided by third-party driver developers, should be installed in:

`/etc/openwin/lib/xil/devhandlers`

Note – The environment variable `XILHOME` no longer exists and will not be read by XIL. If the device handlers are not in `/etc/openwin/lib/xil/devhandlers`, they will not be found.

Compute devices are identified in the `OWconfig` file. See Chapter 4, “Compute Devices,” for more information.

Note – The file `xil.compute` is no longer used as a configuration file for handlers and their dependencies. Instead, compute devices are identified in the `OWconfig` file.

Note – Be sure not to overwrite any existing files when you write your device handlers to the `devhandlers` directory.

Error Reporting for XIL Device Handlers

All the possible error messages in the XIL library are listed in Appendix B, “XIL Error Messages,” in the *XIL Programming Guide*.

Where possible, you should make use of the currently existing error messages. When you need to use device-specific error messages that are to be included in the standard XIL release, you should create a new error file. The *XIL Programming Guide* contains the device-independent error message IDs. These IDs are numbered and prefixed with the string `di-` (for example, `di-312`).

For device-dependent errors, the prefix for the error ID should be the device name for the handler. For example, for a handler with the device name *XXX*`Camera` (where *XXX* is the company name), the error IDs should have the form *XXX*`Camera-123`. In this example, the XIL library looks for the device-specific error message number `123` in the directory

`/usr/openwin/lib/xil/locale/`*current_locale*`\`
`/LC_MESSAGES/`*XXX*`Camera.mo`

The XIL library is internationalized; that is, it uses functions to extract error messages for a given locale. For information on localization of error messages, see the document *Developer's Guide to Internationalization* (available in AnswerBook).

Version Control for XIL Handlers

The XIL core contains the global function:

```
xilVersionPtr* XilGetVersion()
```

This function returns a pointer to a structure that contains 16-bit unsigned integers containing the major and minor release numbers of the current XIL library. The structure looks like this:

```
typedef struct {
    Xil_unsigned16  majorVersion;
    Xil_unsigned16  minorVersion;
}   *xilVersionPtr;
```

The rules for loading handlers are fairly simple:

- The library will not load a module with a `majorVersion` greater than its own. An attempt to load a module greater than the current library version results in an error.
- Currently, the allowable (earlier) module versions that are supported are versions 1.1 and 1.2. Thus, `majorVersion` can only equal 1, and `minorVersion` can equal either 1 or 2.
- The library loads and executes any module with the same `majorVersion` number.

Similar version control rules exist for all of the OGI foundation libraries, including the port for the OpenWindows software.

These rules have implications for writers of XIL device handlers. You should write your handler with the earliest version of the Solaris operating system that you wish to support. Upgrading to a new operating system version by the end user will, in general, not require a new release of XIL device handlers. If you wish to write a handler that requires functionality only available after a specific library release, you must check the `majorVersion` and `minorVersion` numbers to make sure the handler has been loaded by an appropriate version of the library.

For the XIL library to properly load handlers, the name of the handler must contain its major version number as a suffix. For example, the standard XIL I/O handler for X11 support is called `xilioxlib.so.1`. For the 1.*x* release of the XIL library, it is sufficient to ensure that each handler name includes the suffix `.1`.

How XIL Device Handlers Work

Each type of device in the XIL library handles a different aspect of imaging device dependence. The inner workings of each type of device are detailed in Chapter 3, “I/O Devices,” through Chapter 6, “Storage Devices,” in this guide, along with pointers to examples of each device handler. However, the overall concept behind providing a device handler is similar among different kinds of devices.

Note – If you are writing a device handler, be aware that not all XIL application programs call `xil_close()` before exiting. Therefore you should make sure, if possible, that your device handler releases any persistent system resources if an application dies abnormally.

To implement a specific device, you must define a derived class from the appropriate `XilDeviceManager` class that represents the device. Only one derived class can exist for each device, and therefore only one for each handler. The purpose of the derived class is to:

- Initialize the device.
- Create the derived `XilDevice` class.

As an example, consider the case of an I/O device called *XXX*`Camera`, which represents the combination of a frame grabber and camera (as shown in Figure 2-1).

Note – The *XXX* in the device name represents the name of the company creating the device. For details on I/O device naming, see "Adding an I/O Device" on page 91.

This example demonstrates the flow of creating an XIL handler, as follows:

1. The subclass `XilDeviceManagerIO`*XXX*`Camera` is created by a call to the `XilDeviceManager` member function `create()`, which must exist in the loadable library that contains the handler.
2. The `XilDeviceManagerIO`*XXX*`Camera` subclass initializes the frame grabber, and holds all the global information that is shared among different instances of the actual device. The `create()` function is called when the handler is loaded. In this example of an I/O device, this happens the first *XXX*`Camera` handler name as the device-name parameter.
3. After initializing, the XIL core code calls code in `XilDeviceManagerIO`*XXX*`Camera` that creates an instance of the derived class `XilDeviceIO`*XXX*`Camera`. This class contains all the code needed to perform the image acquisition. The second time the application calls `xil_create_from_device()` with the same device name, the second instance of `XilDeviceIO`*XXX*`Camera` is created. To the application, this appears as a second device image. The two device images can exist in sequence or simultaneously.

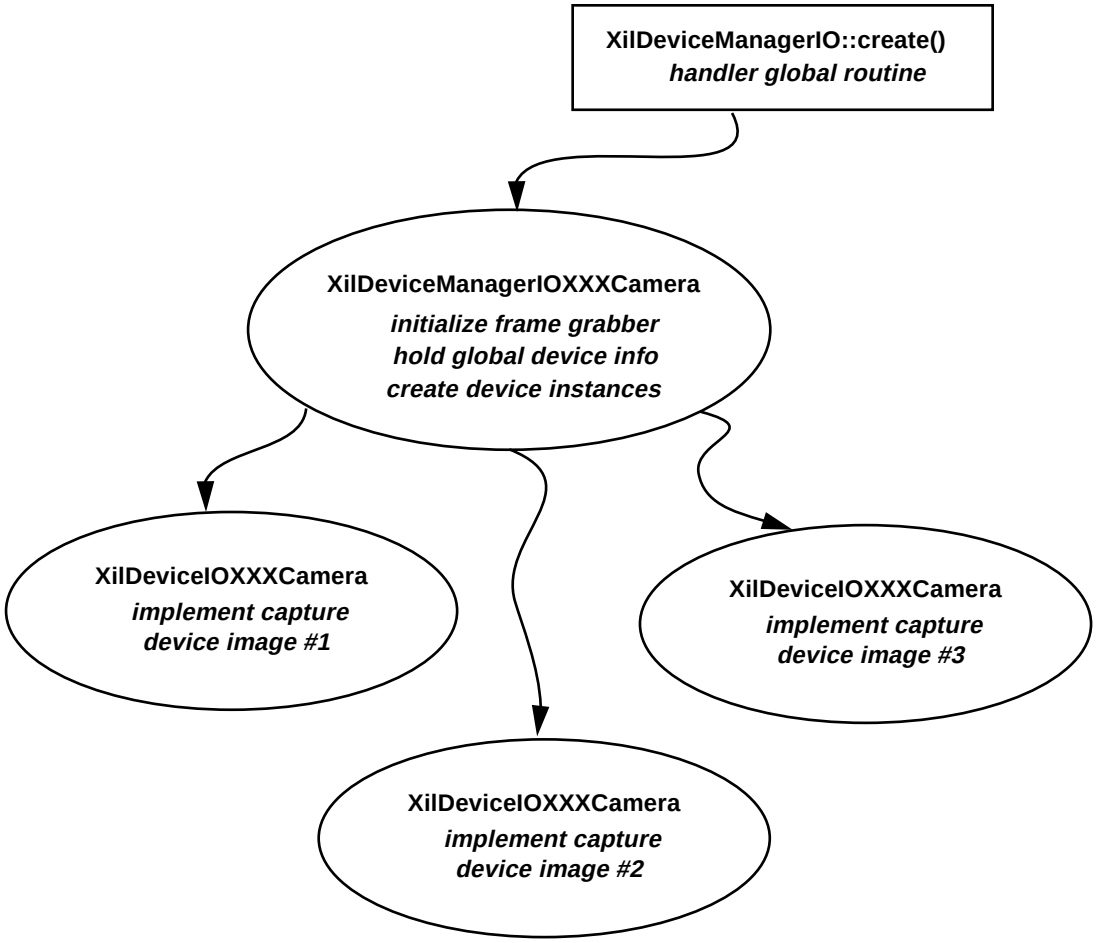


Figure 2-1 An Example of Creating an I/O Handler

The flow of creating a device handler is essentially the same for I/O, storage, and compression handlers (Figure 2-2), but is not identical for compute devices.

Figure 2-2 Flow of Creating an I/O, Storage, or Compression Handler

Compute devices have only a single instantiation, which is controlled by the XIL core code. Thus, there is no derived class called `XilDeviceCompute`; only the `XilDeviceManagerCompute` class exists. Like the other device classes, the `XilDeviceManagerCompute` class must be subclassed, this time to represent the XIL functions that are being accelerated. The mechanism for allowing the XIL core code to instantiate the compute class is described in detail in Chapter 4, "Compute Devices."

Implementing an XIL Operation

To create a handler for a compute, I/O, or compression device, you must implement an XIL operation.

An XIL operation is a member of one of the following classes:

- `XilDeviceManagerCompute`
- `XilDeviceManagerIO`
- `XilDeviceManagerCompression`

Note – This version of the XIL Graphics Porting Interface (GPI) is different from earlier versions: you may now implement XIL operations as part of an I/O, compression, or compute device.

While each type of handler has its own unique features and capabilities, implementing an XIL operation as a member of a compute, I/O, or compression class is essentially the same.

Operation Prototype: Atomic Function

This section specifically looks at code that implements a compute routine for the byte case of the `add` atomic function. However, you can generally apply the basic structure it describes to atomic functions for I/O and compression devices.

For a list of all the atomic functions that you can implement, see Appendix B, "XIL Atomic Functions."

The prototype for the add operation is shown below.

```
int
XilDeviceManagerComputeBYTE::Add(
    XilOp*      op,          // Pointer into the DAG
    int         op_count,    // Number of combined ops to be done
    XilRoi*     roi,         // Region of interest used in the op
    XilBoxList* bl)          // List of boxes to be processed
```

As you recall, the `XilOp` class holds the information required to store an XIL operation in a Directed Acyclic Graph (DAG). The operation parameters are `op`, `op_count`, `roi`, and `bl`.

For an XIL atomic function, the `op` parameter is a pointer to the `XilOp` object that represents the specific atomic function in the DAG. The source and destination images must be accessible to the function. These images are stored in the `XilOp` object. The parameters of the atomic function also are stored in the `XilOp` object. The `XilOp` class contains member functions that enable you to extract the image and parameter information for the atomic function. Appendix A, "XilOp Object," identifies the images and parameters supported by each XIL atomic function and explains how to extract them.

The `op_count` parameter is the number of operations (also called *ops*) combined in the operation. For an atomic function, this number is 1. (For a molecule, the value is a number greater than 1.)

The `roi` parameter is a pointer to the `XilRoi` object, which stores the intersected region of interest (ROI) for the destination image.

Note – In this version of XIL, the compute routine is not required to calculate the intersected ROI while processing. The core has already done this.

The intersected ROI represents the overlapping regions of all source images and the destination image with image origins aligned. The intersected ROI represents that portion of the destination to be written.

The `bl` parameter is a box list. The *box list* is a list of destination areas and their corresponding source areas to be processed by the operation.

A box list may have more than one entry. Each entry contains one box representing a portion of the destination image to be processed in an operation. In addition, the entry contains a box corresponding to each source image. The boxes are used to acquire the storage for each image. Before processing, the ROI is intersected with the box to produce the rectangles relative to the box that need to be processed.

Basic Structure: Atomic Function

This section looks at the code used to implement a compute routine for the `byte` case of the `add` atomic function.

The complete source code for the `XilDeviceManagerComputeBYTE::Add()` atomic function example can be found in the TBD directory.

The basic structure consists of the following steps:

1. **Split boxes in the box list on tile boundaries in the sources.**
2. **Obtain the necessary images and `XilOp` object parameters.**
3. **Loop over the boxes to account for all boxes in the box list.**
4. **Acquire storage for each box.**
5. **Process the data.**

Note – Although this section describes a compute routine, all compute, I/O, and compression routines have this same basic structure. This was not true for previous versions of XIL.

Step 1: Splitting Boxes on Tile Boundaries

Note – Generally the boxes passed in to an operation are already split for tiles in the destination image so the image does not span tiles. (Some exceptions are noted in specific sections of Chapter 4, “Compute Devices.”) Boxes have not yet been split for tiles in the source image.

The first step of the basic structure has the operation take the box list and split boxes on tile boundaries for the source images, as shown below.

```
XilStatus
XilDeviceManagerComputeBYTE::Add(XilOp*       op,
                                 unsigned int   ,
                                 XilRoi*      roi,
                                 XilBoxList*  bl)
{
    if(op->splitOnTileBoundaries(bl) == XIL_FAILURE) {
        return XIL_FAILURE;
    }
```

This step guarantees that the storage you later acquire for the boxes (see “Step 4: Acquiring Storage”) lies within a tile. Because this step involves an operation-specific call, you should see Chapter 4, “Compute Devices,” for potential specific information.

Obtaining Necessary Images and `XilOp` *Object Parameters*

The example code below shows how to get the images for the operation and the number of bands in the image. See Appendix A, “XilOp Object,” for a summary of the parameters available for each operation and a description of the `XilOp` member functions you need to call to obtain the parameter information.

```
XilImage* src1 = op->getSrcImage(1);
XilImage* src2 = op->getSrcImage(2);
XilImage* dest = op->getDstImage(1);

unsigned int nbands   = dest->getNumBands();
```

Step 3: Looping Over Boxes

The compute routine iterates over the box list until no there are no more storage boxes to be processed. Each box in the list describes the area required to perform the image operation. While it is possible to retrieve the coordinates of the image area from the box, most operations simply pass the boxes retrieved from the list as arguments to other functions.

Each entry in a box list contains a box for each image required by that operation. As an example, a box list entry for the add operation consists of three boxes: one for each of the two source images and one for the destination image.

In the example below, `XilBox::getNext()` returns `FALSE` when all boxes have been processed.

```
    XilBox* src1_box;
    XilBox* src2_box;
    XilBox* dest_box;
    while(bl->getNext(&src1_box, &src2_box, &dest_box)) {
```

Step 4: Acquiring Storage

The next step of the basic structure acquires storage for each box. Before describing this procedure, you should understand what `XilStorage` objects are and how to construct them.

`XilStorage` ***Object***

The `XilStorage` object represents the description of a contiguous region of memory associated with a given image. At construction, they contain no information but are filled in by subsequent calls. For improved performance and to ensure that they are destroyed automatically at the end of the loop, `XilStorage` objects should be constructed on the stack as shown below.

```
XilStorage  src1_storage(src1);
XilStorage  src2_storage(src2);
XilStorage  dest_storage(dest);
```

Filling in the `XilStorage` *Object*

You fill in the `XilStorage` objects for the given boxes by calling `XilImage::getStorage()` for the appropriate images, as shown below.

```
if((src1->getStorage(&src1_storage, op, src1_box, "XilMemory",
                              XIL_READ_ONLY)  == XIL_FAILURE) ||
   (src2->getStorage(&src2_storage, op, src2_box, "XilMemory",
                              XIL_READ_ONLY)  == XIL_FAILURE) ||
   (dest->getStorage(&dest_storage, op, dest_box, "XilMemory",
                              XIL_WRITE_ONLY) == XIL_FAILURE)) {
    //
    //  Mark this box entry as having failed.  If marking the box
    //  returns XIL_FAILURE, then we return XIL_FAILURE.
    //
    if(bl->markAsFailed() == XIL_FAILURE) {
        return XIL_FAILURE;
    } else {
        continue;
        }
}
```

The first parameter to `getStorage()` is the address of the `XilStorage` object to be filled.

The second parameter is the `op` that was passed in to this function.

Note – The `op` passed to the `getStorage()` call is different for molecules. See "Operation Prototype: Molecule" on page 62 for more information.

The third parameter is the name of the storage that is being requested. The default is `XilMemory`. If, however, you have your own storage device, this is where you would specify its name.

The fourth parameter is the `XilStorageAccess` type. Source images should always be accessed as `XIL_READ_ONLY`. Destination images are accessed as either `XIL_WRITE_ONLY` or `XIL_READ_WRITE` depending on whether the destination can be read from as well as written to (as is the case with the `xil_copy_with_planemask()`function, or for some optimizations).

Note – It is very important that you request the storage with the correct `XilStorageAccessType`. Not doing so may cause the XIL core to incorrectly prepare the storage and may generate incorrect results.

Two additional parameters need not be specified as they are default parameters. These are an `XilStorageType` and a `void*`.

The fifth parameter, an `XilStorageType`, has the default value of `XIL_STORAGE_TYPE_UNDEFINED`, which means that any of XIL's supported storage types are acceptable and can be processed. If the operation wants to restrict its processing to another type such as `XIL_PIXEL_SEQUENTIAL`, it specifies that as an argument.

Note – It is strongly recommended that you use the default value for `XilStorageType`. Unless you know exactly what you are doing, specifying a different value could significantly impede performance.

The sixth parameter, a `void*`, allows storage-device specific attributes to be passed in by the compute routine. This parameter would only be used if you wrote your own storage device that needed additional information. The `XilMemory` storage device in the example shown takes no additional attributes. However, if you write your own storage device, you may need to pass in additional information.

Note – Failure can be indicated on a per-box basis. If one box can't be acquired or processed, the operation simply marks it as failed and continues. The XIL memory code picks up that portion of the operation that remains undone.

Determining Storage Formats of the Images

Once storage has been acquired from the images, the compute routine can then determine the storage format and set up its processing loop accordingly. XIL supports three storage layouts:

- `XIL_PIXEL_SEQUENTIAL`
- `XIL_BAND_SEQUENTIAL`
- `XIL_GENERAL`

Although the memory compute routines handle all three storage formats, the compute routine may want to restrict its processing to certain types or to write a different image processing loop for each format. For more information on storage formats, see Chapter 4, “XIL Storage,” in the *XIL Programmer’s Guide.*

```
if((src1_storage.isType(XIL_PIXEL_SEQUENTIAL)) &&
   (src2_storage.isType(XIL_PIXEL_SEQUENTIAL)) &&
   (dst_storage.isType(XIL_PIXEL_SEQUENTIAL)) ) {
```

Getting Storage Information

To obtain information from a storage object such as the starting address of the image data or the pixel stride, the `XilStorage::getStorageInfo()` is called. These examples are two overloaded methods for obtaining this data.

```
// Pixel Sequential
unsigned int   src1_pixel_stride;
unsigned int   src1_scanline_stride;
Xil_unsigned8* src1_data;
src1_storage.getStorageInfo(&src1_pixel_stride,
                            &src1_scanline_stride,
                            NULL, NULL,
                            (void**)&src1_data);

// Band Sequential, General
for(unsigned int band=0; band<nbands; band++) {
    unsigned int   src1_pixel_stride;
    unsigned int   src1_scanline_stride;
    Xil_unsigned8* src1_data;
    src1_storage.getStorageInfo(band,
                                &src1_pixel_stride,
                                &src1_scanline_stride,
                                NULL,
                                (void**)&src1_data);
```

For details on storage formats, see Chapter 4, “XIL Storage,” in the *XIL Programmer’s Guide.*

Step 5: Processing the Data

At this point the basic compute routine is ready to process the data.

Obtaining Intersected ROIs

To account for any ROIs that may be set on the images, the routine creates an `XilRectList` object using the `roi` passed in and the destination box. The `XilRectList` constructor gets a list of rectangles to be processed on the destination. Passing in the box ensures that the rectangles lie within the box area and causes the rectangles to be relative to the *starting x and starting y* of the box.

The ROI passed in to the compute routine represents the intersected ROI of the operation. (Any deviations from this are noted in the specific compute device sections in Chapter 4, “Compute Devices.”) The XIL core has correctly mapped and intersected the source and destination ROIs as appropriate for the operation.

The code fragment shows the `XilRectList` object being created on the stack. This enhances performance, as it eliminates the overhead of using `new()` and `delete()` functions.

```
XilRectList    rl(roi, dest_box);

   int            x;
   int            y;
   unsigned int   xsize;
   unsigned int   ysize;
   while(rl.getNext(&x, &y, &xsize, &ysize)) {
     ...
```

Note – The compute routine can construct an `XilConvexRegionList` instead of an `XilRectList`. An `XilConvexRegionList` is used primarily in `affine()` and `rotate()` geometric operations. For details, see Chapter 4, “Compute Devices.”

Using the Appropriate X an d Y Values

With the `XilRectList` object created, the compute routine then moves to the location of the data to start processing using the `x` and `y` values, and it processes data until all the pixels in the rectangles have been dealt with.

Note – The `x` and `y` locations are relative to the *current box* being processed (that is, they are not image coordinates). The data pointer returned from the storage object also is relative to the current box.

The following code shows how to calculate the appropriate data starting point for an `XIL_PIXEL_SEQUENTIAL` byte image.

```
XilRectList    rl(roi, dest_box);
int            x;
int            y;
unsigned int   xsize;
unsigned int   ysize;

while (rl.getNext(&x, &y, &xsize, &ysize)) {
    Xil_unsigned8* src1_scanline = src1_data +
        (y*src1_scanline_stride + (x*src2_pixel_stride);

    Xil_unsigned8* src2_scanline = src2_data +
        (y*src2_scanline_stride + (x*src2_pixel_stride);

    Xil_unsigned8* dest_scanline = dest_data +
        (y*dest_scanline_stride + (x*dest_pixel_stride);
```

Handling Failure and Return Values

As mentioned in “Step 4: Acquiring Storage” on page 56 and “Step 5: Processing the Data” on page 59, failure can be indicated on a per-box basis. This allows the operation to continue looping through the box list, processing those boxes that it can. If any of the boxes has been marked as failed, it is best to return `XIL_FAILURE` at the completion of the whole operation. Although the XIL core catches failed boxes (even if the routine returns `XIL_SUCCESS`), it is more efficient to indicate the failure. If all boxes are processed successfully, the operation returns `XIL_SUCCESS` upon completion of the box list loop.

Note – Because the `XilStorage` object and `XilRectList` are created on the stack, they are destroyed automatically at the end of each loop through the box list. Since the information in them is specific to the boxes being processed

during any given loop, there is no reason to make them persistent. If you do not create them on the stack, you must explicitly destroy them upon completion of looping through the box list.

Operation Prototype: Molecule

The prototype for a molecule looks exactly like that for an atomic function. (Compare the prototype in "Operation Prototype: Atomic Function" on page 53 to the molecule prototype shown here.)

```
int
XilDeviceManagerComputeBYTE::ThresholdThreshold(
    XilOp*      op,         // Pointer into the DAG
    int         op_count;   // Number of combined ops to be done
    XilRoi*     roi;        // Region of interest used in the op
    XilBoxList* bl)         // List of boxes to be processed
```

As in an atomic function, the `XilOp` class holds all the information required to store an XIL operation in the DAG. In the case of a molecule, however, the `op` parameter is a pointer to the `XilOp` object that represents the last in a sequence of XIL operations in the DAG. The `op_count` indicates the number of XIL operations combined in the molecule.

Any of the previous operations in the molecule's operation sequence are accessible from the passed-in `op` parameter. The source, destination, and parameters for each individual `op` in the sequence are available to you from the appropriate `XilOp` object.

As in an atomic function, the `roi` parameter is a pointer to the `XilRoi` object, which stores the intersected ROI for the destination image. In the case of a molecule, the intersected ROI represents the intersected ROI that would be generated by doing the individual operations in sequence from the start of the molecule to the final destination.

Note – There are restrictions on which operations are deferrable in XIL1.3. For more information, see "Some Considerations" on page 32.

The `bl` parameter is the box list. The box list is a list of destination areas and their corresponding source areas to be processed by the operation. In the case of a molecule, the source areas referenced are those for the sources to the first `XilOp` object in the operation sequence. The destination areas refer to the destination of the last `XilOp` object in the operation sequence. Secondary sources on intermediate ops would follow in order between the initial sources and the final destination.

Basic Structure: Molecule

This section looks at the code used to implement a compute routine for the byte case of the `threshold-threshold` molecule.

The complete source code for the `XilDeviceManagerComputeBYTE::ThresholdThreshold()` molecule example can be found in the TBD directory.

The structure of a molecule is very similar to that for an atomic function. The basic structure consists of the following steps:

1. **Optionally verify which molecule is being handled.**
2. **Obtain the individual `XilOp` objects from the `op` parameter.**
3. **Split boxes in the box list on tile boundaries in the source.**
4. **Obtain the images and `XilOp` object parameters.**
5. **Loop over the boxes to account for all boxes in the box list.**
6. **Acquire storage for the boxes using the appropriate `XilOp` objects.**
7. **Process the data.**

This basic structure differs from the atomic function basic structure (see “Basic Structure: Atomic Function” on page 54) in the following ways:

- The first two (additional) steps apply to molecules only.
- Step 3 (splitting boxes on tile boundaries) and step 6 (acquiring storage) include some modifications for molecules.

Step 1: (Optional) Verifying the Passed-In Molecule

The XIL core ensures that a molecule does not get called unless it meets the criteria defined when the molecule is registered. It is not necessary for the molecule to check. Checking the `op_count` can be valuable, however, when the function implements more than one molecule. In such cases, the `op_count` indicates which molecule is being called. Because only one molecule is implemented in this example, `op_count` does not require verification.

```
XilStatus
XilDeviceManagerComputeBYTE::ThresholdThreshold(XilOp*      op,
                                                unsigned    op_count,
                                                XilRoi*     roi,
                                                XilBoxList* bl)
{
```

A common example of a function that would handle molecules of different lengths is a decompressed-display molecule that has an optional extra copy.

Step 2: Obtaining the `XilOp` *Objects and Their Parameters*

The `op` parameter contains a pointer to the list of the deferred operations in depth-first order (that is, the first `XilOp` object in the list is the bottom `op` in the operation sequence). The compute routine views all the individual `op`s in a molecule as one operation. As such, it is only interested in the destination image from the final `XilOp` object in the `op` sequence.

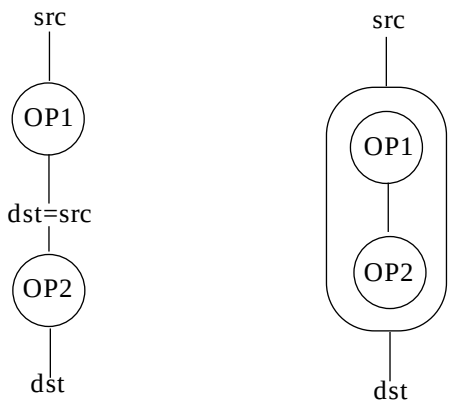


Figure 2-3 Operation Sequence

As shown in the code example below, the `op` parameter is the same as the 0 entry in the `op` list (the first position in the list). The first `XilOp` object in the sequence of operations is always in the position (`op_count` -1) in the `op` list.

```
//
//  This molecule only has two XilOp objects. The top op
//  provides the source image. The bottom op provides the
//  destination image.
//
XilOp* src_op = op->getOpList()[1];
XilOp* dst_op = op;

if(dst_op->splitOnTileBoundaries(bl) == XIL_FAILURE) {
    return XIL_FAILURE;
}
```

Step 3: Splitting Boxes on Tile Boundaries

This next step has the operation take the box list and split boxes on tile boundaries for the source images, as shown below. Although this code could be exactly the same as the atomic function code for this step, it is important to realize that the `XilOp::splitOnTileBoundaries()` function is being called on the destination op.

```
{
    if(dst_op->splitOnTileBoundaries(bl) == XIL_FAILURE) {
        return XIL_FAILURE;
    }
```

Step 4: Obtaining Images and `XilOp` *Object Parameters*

The molecule function needs to get the source images from the first `op` in the operation sequence, the destination image from the last `op` in the operation sequence, as well as any additional source images needed by intermediate `ops` in the sequence. (See Figure 2-3 on page 65.) Once the `XilOp` objects are obtained (see “Step 2: Obtaining the XilOp Objects and Their Parameters” on page 64), you use the same functions to obtain the parameters from each

individual op as you would for an atomic function, as shown in the code example below. For more information on these functions, see Appendix A, “XilOp Object.”

```
XilImage* src1_image = src_op->getSrcImage(1);
XilImage* dest_image = dst_op->getDstImage(1);

//
//  First Threshold
//
Xil_unsigned8* op1_low;
src_op->getParam(1, (void**)&op1_low);

Xil_unsigned8* op1_high;
src_op->getParam(2, (void**)&op1_high);

Xil_unsigned8* op1_map;
src_op->getParam(3, (void**)&op1_map);

//
//  Second Threshold
//
Xil_unsigned8* op2_low;
dst_op->getParam(1, (void**)&op2_low);

Xil_unsigned8* op2_high;
dst_op->getParam(2, (void**)&op2_high);

Xil_unsigned8* op2_map;
dst_op->getParam(3, (void**)&op2_map);

//
//  Store away the number of bands for this operation.
//
unsigned int num_bands   = dest_image->getNumBands();
```

Step 5: Looping Over Boxes

The compute routine now iterates over the box list until no there are no more storage boxes to be processed. This step is exactly the same as if this were an atomic function.

```
    XilBox* src1_box;
    XilBox* dest_box;
    while(bl->getNext(&src1_box, &dest_box)) {
```

Step 6: Acquiring Storage

The next step of the basic molecule acquires storage for the boxes. This differs from an atomic function in that the `op` parameter passed in to the `XilImage::getStorage()` call for a given image must match the `XilOp` object with which the image is associated. As shown below, the source image comes from the top operation (`src_op`), and the destination image comes from the bottom operation (dst_op).

```
//
//  Aquire our storage from the images.  The storage returned is valid
//  for the box given.  Thus, any origins or child offsets have been
//  taken into account.
//
XilStorage src1_storage(src1_image);
XilStorage dest_storage(dest_image);
if((src1_image->getStorage(&src1_storage,src_op, src1_box,
       "XilMemory", XIL_READ_ONLY) == XIL_FAILURE) ||
   (dest_image->getStorage(&dest_storage, dst_op, dest_box,
       "XilMemory", XIL_WRITE_ONLY) == XIL_FAILURE)) {
    //
    //  Mark this box entry as having failed.  If marking the box
    //  returns XIL_FAILURE, then we return XIL_FAILURE.
    //
    if(bl->markAsFailed() == XIL_FAILURE) {
        return XIL_FAILURE;
    } else {
        continue;
        }
}
```

All other parameters to the `XilImage::getStorage()` call are the same as those for an atomic function.

Step 7: Processing the Data

Once storage has been acquired from the images, the compute routine can then determine the storage format and set up its processing loop accordingly.

The first part of this routine shown here is specialized for `XIL_PIXEL_SEQUENTIAL` data.

```
//
//  Test to see if all of our storage is of type XIL_PIXEL_SEQUENTIAL.
//  If so, implement a loop optimized fro pixel-sequential storage.
//
if((src1_storage.isType(XIL_PIXEL_SEQUENTIAL)) &&
   (dest_storage.isType(XIL_PIXEL_SEQUENTIAL))) {
    unsigned int   src1_pstride;
    unsigned int   src1_sstride;
    Xil_unsigned8* src1_data;
    src1_storage.getStorageInfo(&src1_pstride, &src1_sstride,
        NULL, NULL,(void**)&src1_data);

    unsigned int   dest_pstride;
    unsigned int   dest_sstride;
    Xil_unsigned8* dest_data;
    dest_storage.getStorageInfo(&dest_pstride, &dest_sstride,
        NULL, NULL,(void**)&dest_data);

    //
    //  Create a list of rectangles to loop over.  The resulting list
    //  of rectangles is the area left by intersecting the ROI with
    //  the destination box.
    //
    XilRectList    rl(roi, dest_box);

    int            x1;
    int            y1;
    unsigned int   xsize;
    unsigned int   ysize;
    while(rl.getNext(&x1, &y1, &xsize, &ysize)) {
```

The latter part of the above routine is shown here. It handles any general storage layout.

```
} else {
    //
    //  For XIL_GENERAL and XIL_BAND_SEQUENTIAL images
    //
    XilRectList  rl(roi, dest_box);

    int          x1;
    int          y1;
    unsigned int xsize;
    unsigned int ysize;
    while(rl.getNext(&x1, &y1, &xsize, &ysize)) {
        //
        //  Each Band...
        //
        for(unsigned int band=0; band<num_bands; band++) {
            unsigned int   src1_pstride;
            unsigned int   src1_sstride;
            Xil_unsigned8* src1_data;
            src1_storage.getStorageInfo(band,
                                        &src1_pstride,
                                        &src1_sstride,
                                        NULL,
                                        (void**)&src1_data);

            unsigned int   dest_pstride;
            unsigned int   dest_sstride;
            Xil_unsigned8* dest_data;
            dest_storage.getStorageInfo(band,
                                        &dest_pstride,
                                        &dest_sstride,
                                        NULL,
                                        (void**)&dest_data);
```

Supporting Re-entrancy

It is critical and expected that routines be fully re-entrant. This is because any operation can be processing any number of operations simultaneously. Furthermore, multiple portions of the same operation can be processed simultaneously.

XIL classes and objects are not multi-thread safe (MT-safe). As such, only acquiring information is allowed. Global or static variables cause the operation to fail.

Pre-Process and Post-Process Methods

Some operations such as lookup may need to pre-calculate information for use by each of multiple threads executing that operation at the same time, and then perform a single cleanup function. To facilitate this, compute, I/O, and compression routines can have optional pre-process and post-process methods. These methods are guaranteed to be called once per operation, but many threads may be calling them at the same time.

Pre-Process Method

The pre-process routine is guaranteed to be called prior to any threads calling the main function. (The post-process routine is not called until all the threads have completed the main function.)

Pre-Process Example

In this example, the pre-process routine fills in a pointer `compute_data` to the data to be shared between compute routines.

```
XilStatus
XilDeviceManagerComputeBYTE::Lookup8Preprocess(XilOp*           op,
                                               unsigned          ,
                                               XilRoi*           ,
                                               void**   compute_data
                                               unsigned int*       )
{
    XilImage* dst = op->getDstImage(1);

    //
    // Create a lookup structure no matter what
    //
    LookupData* lud = new LookupData;
    if(lud == NULL) {
        XIL_ERROR(dst->getSystemState(), XIL_ERROR_RESOURCE, “di-1”, TRUE);
        return XIL_FAILURE;
    }
     ...
```

```
XilStatus
    // Code deleted for brevity
     ...
    *compute_data = lud;

    return XIL_SUCCESS;
}
```

In the example, the `unsigned int` parameter to `Lookup8Preprocess()` is an optional parameter. By default its value is 0. You can use this parameter, for example, if you implement multiple versions of the same function in a given compute routine. You assign a unique value to it for each function so the XIL core can keep track of which version is being associated with a particular pre-process routine.

If the pre-process routine returns `XIL_FAILURE`, the function is never called for this operation. This can be helpful for detecting operation-specific conditions (such as only processing one-banded images) without requiring the main function to test the image format each time it is called.

Accessing the Pre-process Data

The compute routine gets at the pre-process data using a method on the op `XilOp::getPreprocessData()` as shown here.

```
LookupData*  lud = (LookupData*)op->getPreprocessData(this,id);
```

The `id` parameter is optional. It is a unique value passed in to identify this particular version of a function.

Post-Process Method

The post-process method is called after all the compute (I/O or compression) operations have been called. Like the pre-process method, the post-process method is called once per operation but may be called by multiple threads. This method typically would be used to deallocate the pre-process data. The following is an example.

```
XilStatus
XilDeviceManagerComputeBYTE::Lookup8Postprocess(XilOp*            ,
                                                void*  compute_data)
{
    LookupData* lud = (LookupData*)compute_data;

    if(lud->allocated != 0) {
        delete lud;
    }
    return XIL_SUCCESS;
}
```

Registering Operations With the XIL Library

When a compute, I/O, or compression device handler is loaded into the XIL library, it calls the device manager's `describeMembers()` pure virtual function. Your device handler is required to implement this function to describe its capabilities to the XIL library. If `describeMembers()` returns `XIL_SUCCESS`, the library assumes all your device manager functions have been successfully described, and it considers your device handler to have been loaded. If, however, `describeMembers()` returns `XIL_FAILURE`, the library does not consider your device handler loaded and will not use functions from the device.

Generating `describeMembers()`

To simplify generating the `describeMembers()` function—since the GPI to describe functions to the XIL library (particularly molecules) is often a place where mistakes can be made—SunSoft provides the Perl script `xilcompdesc.pl`. You can use `xilcompdesc.pl` to generate a `describeMembers.cc` file from your source files. Then compile `describeMembers.cc` into your device handler.

You invoke the script with three arguments, using the syntax shown here.

```
xilcompdesc.pl <className> <classType> <files>
```

<*className*> is the name of your derived class.

<*classType*> is the type of class (that is, compute, I/O, or compression).

<*files*> is a space-delimited list of files from which to extract `XILCONFIG` information.

This example invokes `xilcompdesc.pl` for a compute handler. `XILCONFIG` information is extracted from three files: `Add.cc`, `Lookup.cc`, and `Multiply.cc`.

```
xilcompdesc.pl XilDeviceManagerComputeMyDevice Compute Add.cc Lookup.cc Multiply.cc
```

Usually, you would incorporate an `xilcompdesc.pl` script into your `Makefile` to generate `describeMembers.cc` and to compile `describeMembers.cc` into your device handler.

`XilConfig` *Syntax Describing an Operation*

The `xilcompdesc.pl` script looks for a well-defined `XILCONFIG` line and an adjoining block describing the member functions implementing an operation in the given source files. The `XILCONFIG` syntax is shown here.

```
//
//  XILCONFIG:  <Member Function>  {
//     OP=<XIL Operation Name>
//     PRE=<Preprocess Member Function>
//     POST=<Postprocess Member Function>
//  }
//
```

<*Member Function*> is the name of the member function in your derived class that implements the functionality described in the adjoining block.

<*XIL Operation Name*> is the GPI name of the XIL operation you're implementing. See Appendix A, “XilOp Object,” for a list of the available atomic function names.

<*Preprocess Member Function*> and <*Postprocess Member Function*> are the names of the member functions in your derived class that act as pre-processor and post-processor routines, respectively, for the primary member function. See “Pre-Process Method” on page 71 and “Post-Process Method” on page 73 for more information on these routines.

In the above syntax, do not precede or follow the equal sign (=) of a description assignment with a space character delimiter. `PRE` and `POST` assignments are optional, but `OP` is required.

Example of Generating `describeMembers()`

Say, for example, an atomic function such as `lookup()` looks up `XIL_BYTE` data and outputs `XIL_BIT` data. As another example, a molecule implements an `add()` followed by a `multiply()` function. The `XILCONFIG` lines to describe each of these operations would look something like this.

```
//
//  XILCONFIG:  Lookup  {
//    OP=lookup;8->1
//    PRE=LookupPreprocess
//    POST=LookupPostprocess
//  }
//
//  XILCONFIG:  AddMul  {
//    OP=add;8
//    OP=multiply;8
//  }
//
```

For the previous two `XILCONFIG` lines, the `xilcompdesc.pl` script generates this `describeMembers()` function.

```
XilStatus
XilDeviceManagerComputeMyDevice::describeMembers()
{
    XilFunctionInfo* fi;

    fi = XilfunctionInfo::create();
    fi->describeOp(XIL_STEP, 1, "lookup;8->1");
    fi->setFunction((XilComputeFunctionPtr)
                    XilDeviceManagerComputeMyDevice::Lookup,
                    "lookup;8->1()");
    fi->setPreprocessFunction((XilComputePreprocessFunctionPtr)

XilDeviceManagerComputeMyDevice::LookupPreprocess)
    fi->setPostprocessFunction((XilComputePostprocessFunctionPtr)

XilDeviceManagerComputeMyDevice::LookupPostprocess)
    this->addFunction(fi);
    fi->destroy();

    fi = XilfunctionInfo::create();
    fi->describeOp(XIL_STEP, 1, "multiply;8");
    fi->describeOp(XIL_STEP, 1, "add;8");
    fi->setFunction((XilComputeFunctionPtr)
                    XilDeviceManagerComputeMyDevice::AddMul,
                    "multiply;8(add;8())");
    this->addFunction(fi);
    fi->destroy();

    return XIL_SUCCESS;}
```

Generic Steps To Writing a Device Handler

The following outlines the steps to adding a device handler to the XIL library:

1. **Write the device manager and device class for your driver.**
   See “Implementing an XIL Operation” on page 52.

2. **Determine versioning for your handler.**
   See “Version Control for XIL Handlers” on page 48.

3. **Write the operations (molecules and/or atomic functions).**
   Write the imaging functions you need for your device handler.

4. **Add your handler to the config file.**
   See “Loading Compute Handlers” on page 94.

5. **Register the operations with the XIL Library.**
   See “Registering Operations With the XIL Library” on page 73.

6. **Test using the XIL Test Suite.**
   Test your device handler using the guidelines in the *XIL Test Suite User’s Guide*.

2

I/O Devices 3

This chapter introduces I/O devices and identifies the basic functions required to create them. In addition, the chapter provides a step-by-step procedure for adding an I/O device.

About I/O Devices

In the XIL Imaging Library, I/O devices include any devices that can generate or receive images, such as frame grabbers, image files, and displays. The XIL library supports these types of devices by allowing them to appear as device images to an application. When a device image is used as a source in an operation, an image is captured from the device. When a device image is used as a destination in an operation, an image is written to the device.

The I/O device handler provides an implementation for an image captured from a device and for an image written to a device. The first time a device image is created using the `xil_create_from_device()` API call, the software module containing the handler is loaded. Once the I/O handler is loaded, any compute devices that have only the I/O handler as a dependence

are loaded. The I/O handler information is cached so that subsequent creations of new device images from the same device do not require reloading the I/O handler.

The character string representing the name of the device, passed as the second argument to `xil_create_from_device()`, is used to select the appropriate loadable library. Currently, the following API call attempts to load an I/O handler named `/etc/openwin/lib/xil/devhandlers/xilIO_my_device.so.2` and fails with an error if this loadable library does not exist:

```
device_image = xil_create_from_device(systemState, “my_device”, NULL);
```

I/O Device Capabilities

The current version of the XIL library has added several new capabilities that affect I/O devices:

- Tiling
- Molecules
- Multithread safe (MT-safe)
- Double buffering
- Push devices

Tiling, a feature common to all of XIL as of the current library version, has certain implications for I/O devices. First, an I/O device must be able to provide or process portions of an image as the GPI requests. Second, the I/O device is responsible for constructing the *controlling image* for the device, which is returned to the user.

The library has the ability to implement molecules without a compute routine.

I/O devices are expected to be MT-safe. This means a framebuffer may have to lock its registers so their contents are accessed by one thread at a time when multiple threads are running.

XIL supports devices that use double-buffering. (See “Functions for Double Buffering Devices” on page 90 for more information.)

Finally, a new type of device called a *push* device controls data flow by providing data to the core as it is ready. By contrast, *pull* devices provide data when the core requests it. While a pull device is the more common

implementation, the two methods are not mutually exclusive. When the core requests the data from a push device, the device is expected to fill in the requested area at its own pace and to notify the core when processing of the data is complete. This can be useful for sequential access devices such as scanners. The core guarantees that the area requested is the entire area needed and that no asynchronous requests will be made.

Implementing an I/O Device

To implement an I/O device, it is necessary to implement a device manager as well as a device.

Implementing an I/O Device Manager

The device manager exists from the time the I/O device is created by an API call such as `xil_create_from_device()` until `xil_close()` is called. Only one device manager exists for any single device. Even if you have two GX framebuffers on the system, there is only one device manager. However, that device manager is responsible for creating GX I/O devices on multiple devices

The complete code to the I/O device manager example can be found in the TBD directory.

Creating a Device Manager

To create a device manager, you derive a class from `XilDeviceManagerIO` as shown below. Your device manager is where you might store persistent data that is not instance specific.

```
class XilDeviceManagerIOcg6 : public XilDeviceManagerIO {
public:
...
// class definitions go here
...
}
```

Required Device Manager Functions

You must overload certain functions must be overloaded in your device manager class:

- static `create()`
- one device constructor function
- `getDeviceName()`
- `describeMembers()`

`create()`

The first function you must overload is a static `XilDeviceManagerIO::create()`. This is the entry point for the XIL library.

```
static XilDeviceManagerIO* create(unsigned int libxil_gpi_major,
                                  unsigned int  libxil_gpi_minor,
                                unsigned int* devhandler_gpi_major,
                               unsigned int* devhandler_gpi_minor);
```

The `create()` function lives in every I/O pipeline and constructs a class derived from `XilDeviceManagerIO`. XIL provides the pipeline with the highest major and minor version numbers of the GPI it supports. At the same time, the compute pipeline is expected to provide XIL with the highest version of the GPI it supports. The compute pipeline is expected to fail if the version is not one that is supported by the pipeline, for example, if there is a mismatch in the major version numbers or the minor version is lower than the one required by the pipeline. XIL may decide not to load the pipeline, or it may decide to alter its behavior to support an older version of the interface.

You can use the macro XIL_BASIC_GPI_VERSION_TEST shown below to test the version number. It is defined in _XilGPIDefines.hh.

```
#define
XIL_BASIC_GPI_VERSION_TEST(lib_major,lib_minor,ptr_major,ptr_mi
nor) \
    {                                                   \
        if(lib_major != XIL_GPI_MAJOR_VERSION ||    \
           lib_minor <  XIL_GPI_MINOR_VERSION) {    \
            return NULL;                            \
        } else {                                    \
            *ptr_major = XIL_GPI_MAJOR_VERSION;     \
            *ptr_minor = XIL_GPI_MAJOR_VERSION;     \
        }                                           \
    }
```

The code below shows how to use XIL_BASIC_GPI_VERSION_TEST.

```
XilDeviceManagerCompute*
XilDeviceManagerCompute::create(unsigned int  libxil_gpi_major,
                                unsigned int  libxil_gpi_minor,
                              unsigned int* devhandler_gpi_major,
                              unsigned int* devhandler_gpi_minor)
{
    XIL_BASIC_GPI_VERSION_TEST(libxil_gpi_major,
                               libxil_gpi_minor,
                               devhandler_gpi_major,
                               devhandler_gpi_minor);

    XilDeviceManagerComputeBYTE* device;

    device = new XilDeviceManagerComputeBYTE;

    if(device == NULL) {
        XIL_ERROR(NULL, XIL_ERROR_RESOURCE, "di-1", TRUE);
        return NULL;
    }

    return device;
}
```

Device Constructor

Your derived I/O device manager should overload one or more of the three device `construct*Device()` functions shown here.

```
//
//  Creates a new XilDeviceIO object for a particular device.
//  This construct call is made through
//  xil_create_from_device()
//
virtual XilDeviceIO*   constructFromDevice(XilSystemState state,
                                           XilDevice*      device);
//
//  Creates a new XilDeviceIO object from a display pointer
//  and window.
//  This construct call is made through
//  xil_create_from_window()
//
virtual XilDeviceIO*  constructDisplayDevice(
                                           XilSystemState* state,
                                           Display*        display,
                                           Window          window);

//
//  Creates a new XilDeviceIO object which supports double
//  buffering froma display pointer and window. At construction
//  the device is expected to be referencing the BACK buffer
//  of the two buffers.
//  This construct call is made through
//  xil_create_from_double_buffered_window()
//
virtual XilDeviceIO*

constructDoubleBufferedDisplayDevice(XilSystemState* state,
                                             Display*      display,
                                              Window        window)
```

`getDeviceName()` *and* `describeMembers()`

You are required to implement two other functions for the device manager class: `getDeviceName()` and `describeMembers()`.

The prototype of each function is shown here. For information on `describeMembers()`, see “Registering Operations With the XIL Library” on page 73.

```
//
//  Required function that returns the name of this device.
//
    const char*            getDeviceName();

//
// Describe the functions we implement to the XIL core
//
    XilStatus              describeMembers();
```

The `getDeviceName()` function returns the name of the device (`xilIO_`*return_value*`.so.2`), for example `SUNWcg6`.

Optional Device Manager Destructor

The device manager destructor provides an opportunity to clear any persistent data generated by the constructor. Because a constructor typically is used to open a device, implementation of the destructor is optional.

```
//
// Constructor Destructor
//
                           XilDeviceManagerIOcg6();
                           ~XilDeviceManagerIOcg6();
```

Implementing a Device

Every device image has an `XilDeviceIO` class. The `XilDeviceIO` class is called to implement the display and capture operations. In addition the class may choose to implement certain sequences of operations that involve a display or capture operation as molecules.

Creating a Device

To create a device, you derive a class from `XilDeviceIO`, which is your device.

```
class XilDeviceIOcg6 : public XilDeviceIO {
public:
...
// class definitions go here
...
}
```

Required Device Functions

There are several functions in the `XilDeviceIO` class, some of which must be overloaded and some of which are optional. The following functions must be implemented:

- `constructControllingImage()`
- `setAttribute()`
- `getAttribute()`
- Functions for readable and/or writable devices
- Functions for double buffering devices

`constructControllingImage()`

The *controlling image* is the image that holds the results of the capture and the source of the display. It is requested by the XIL core through the `constructControllingImage()` function.

```
//
// Return the image created by the device to the
// core, which can then attach the device to it.
//
    XilImage*           constructControllingImage();
```

When a device image is used as a source, the library inserts a device-dependent capture into the current operation sequence and returns the controlling image. When a device image is used as a destination, the library

inserts a copy from the source to the controlling image and then inserts a display operation into the current operation sequence. In the current version of the XIL library, the IO device is responsible for creating the controlling image according to the requirements of the device.

`setAttribute()` *and* `getAttribute()`

The `setAttribute()` and `getAttribute()` functions provide a way for a device to set and get device-specific attributes from the API.

```
//
// Set an attribute on the device
//
XilStatus setAttribute(const char* attribute_name,
                       void*       value);

//
// Get an attribute from the device
//
XilStatus getAttribute(const char* attribute_name,
                       void**      value);
```

I/O devices may define attributes that are used to modify or report their behavior. For example, a frame grabber would use attributes to allow the application to select the type of output image or to select which video input to use. A file input device would use attributes to set the path name.

Note – Device image attributes are defined by the port, but some frame-buffer-specific attributes have already been defined for XIL handlers and *must* be supported.

These attributes are listed in Table 3-1.

Table 3-1 Required Frame Buffer Attributes

Attribute	Value
`COLORMAP`	The X colormap of the device image (write only)

Table 3-1 Required Frame Buffer Attributes *(Continued)*

Attribute	Value
WINDOW	The X window (read only)
DISPLAY	The X display (read only)
COLORSPACE	The color space name

Functions for Readable and/or Writable Devices

You must implement two or all four of the following functions depending on whether the device is readable, writable, or both.

- `capture()`and `getPixel()`
- `display()` and `setPixel()`

```
//
// Is the device readable - if it is readable, then
// the capture() and getPixel() functions must be implemented.
//
Xil_boolean isReadable();

//
// Is the device writable - if it is writable, then the
// display() and setPixel() functions must be implemented.
//
Xil_boolean isWritable();
```

If the function `isReadable()` returns `TRUE`, the `getPixel()` and `capture()` functions must be implemented. If the `isWritable()` function returns `TRUE`, the `display()` and `setPixel()` functions must be implemented.

```
//
// Get a pixel from the device
//
XilStatus getPixel(unsigned int x,
                   unsigned int y,
                   float*       data,
                   unsigned int offset_band,
                   unsigned int nbands);
//
// Capture an image from the device
//
XilStatus capture(XilOp*       op,
                  unsigned int op_count,
//
// Display an image on the device
//

XilStatus display(XilOp*       op,
                  unsigned int op_count,
                  XilRoi*      roi,
                  XilBoxList*  bl);
//
// Set a pixel on the device
//
XilStatus setPixel(unsigned int x,
                   unsigned int y,
                   float*       data,
                   unsigned int offset_band,
                   unsigned int nbands);
```

Note – `capture()` would only be implemented if the device is a pull device. For a push device, the class would need to implement `startCapture()` and `stopCapture()`.

Functions for Double Buffering Devices

In addition, for those devices that represent double-buffered devices and whose creation function is `constructDoubleBufferedDisplayDevice()`, you must implement the functions described in Table 3-2.

Table 3-2 Double Buffering Device Functions

Function	Description
`getActiveBuffer()`	Returns the buffer into which data is written
`setActiveBuffer()`	Specifies whether the active buffer is to be set to the front or back
`swapBuffers()`	Moves contents of the back buffer to the front buffer

These functions are shown here.

```
//
//  Set and get the active buffer state for the device
//  and swap back buffer and the front buffer. After a
//  swap the contents of the back buffer are UNDEFINED.
//
//  These are only valid for devices created as a double
//  buffering device via the xil_create_double_buffered_window()
//  call.
//
XilStatus       setActiveBuffer(XilBufferId active_buffer);
XilBufferId     getActiveBuffer();
XilStatus       swapBuffers();
```

Optional Device functions

You may choose to overload the `hasSubPixelCapture()` and `hasSubPixelDisplay()` functions to implement special functionality with your device. These functions may be implemented as a performance optimization when the device allows less than all bands of a pixel to be updated.

For example, by creating a one-banded child of an RGB image, the user may intend on modifying the ‘R’ data only of a display image. If the framebuffer allows for sub-pixel updates, XIL uses the following optimal two-step procedure:

1. **It copies the new ‘R’ band data to the controlling image.**
2. **It copies the ‘R’ band data back to the framebuffer.**

If, however, the framebuffer does not allow for sub-pixel updates, XIL uses a three-step procedure:

1. **It captures all bands to the controlling image.**
2. **It updates the ‘R’ data in the controlling image.**
3. **It copies all three bands back to the framebuffer.**

```
//
//  Indicate whether the device's capture and display routines
//  support sub-pixels.

// This means the capture routine supports using the band_offset
//  num_bands arguments provided.  The default is that routines
//  do not support writing sub portions of pixels to the device.
//
Xil_boolean     hasSubPixelCapture();
Xil_boolean     hasSubPixelDisplay();
```

Adding an I/O Device

Adding an I/O device is straightforward in the XIL library. The handler writer must follow these steps:

1. **Choose a name for your device.**
   It is recommend that you include your company’s stock symbol (if you have one) as part of the name, for example, the `Fred` framebuffer from the company FRED-FX (FFX) would name the IO device `FFXfred`.
2. **Create a device manager class, for example, the class name `XilDeviceManagerIOFFXfred`.**
   Implement the static `create()` function and overload at least one of the constructors and the two required functions, `getDeviceName()` and `describeMembers()`.

3. **Implement the `device` class for the device, for example, the device class name `XilDeviceIOFFXfred`.**
   Implement the required functions `constructControllingImage()`, `setAttribute()`, `getAttribute()`, `isReadable()`, and `isWritable()`. Implement `capture()`, `display()`, and `get/setPixel()` as needed. Implement any of the optional overloaded functions or any molecules as desired.

4. **Identify any new molecules to XIL through use of the `xilcompdesc.pl` script.**
   For pull devices, you must identify `display()` and `capture()` in `describeMembers()`. Push devices do not need to identify `capture()` but must identify `display()`.

5. **Place the new loadable library file in an application package so that it will be installed in the correct location.**
   See the document *SunOS Application Packaging and Installation Guide* for information on using the package system. Also see Chapter 1, "Overview," for information about packaging handlers.

   The name of the loadable library must be unique; it is strongly suggested that you use `xilIO_`*device_name*`.so`, where *device_name* is the name that will be used to describe the device in the `xil_create_from_device()` API call. The `xilIO_` portion of this name is required. As an example, for the device name `FFXfred`, the loadable library name is `xilIO_FFXfred.so`.

Compute Devices 4

About Compute Devices

Compute devices implement XIL image processing operations. The computation can take place on the CPU or on an auxiliary image processing board.

Implementing an XIL Function

Note – The values produced by the implementation of an XIL function should match as closely as possible the values produced by the memory port. This has several implications. Molecules must behave semantically like the sequence of atomic operations, and produce the same (or nearly the same) results as calling the individual atomic functions. A molecule cannot have a greater precision than the atomic functions that the molecule contains. For many of the simple functions, any difference from the default version should not be tolerated. More complicated operations, where there are many floating-point or fixed-point calculations done for each pixel, do not always allow pixel-for-pixel accuracy with any sort of reasonable code. Often, new algorithms provide

slightly different values. It is up to the implementor of the algorithm to make sure that there are no *systematic* differences between the new implementation and the old one.

The XIL Test Suite can aid you in verifying new implementations of XIL functions. The XIL Test Suite enables you to perform regression tests of new code against verified reference signatures and includes the capability of specifying a tolerance for the comparison. This test suite is described in a separate document, *XIL Test Suite User's Guide*, which is part of this release.

Error handling in an implementation is performed by calling the macro `XIL_ERROR`. Both compute handler examples use this interface. The method used to add error messages for device-dependent errors is discussed in Chapter 2, "More on Writing Device Handlers."

Loading Compute Handlers

To provide for maximum flexibility in adding software device handlers, your compute device handler is dynamically loaded as a shared object at run time. The list of loadable objects (device handlers) is maintained in a configuration database file called `config`. You must edit the `config` file using scripts to add entries for compute device handlers that will be used by XIL.

XIL merges the two files from `/etc/openwin/lib/xil/config` and `/usr/openwin/lib/xil/config`. If duplicate information between the two files exists, the information in `/etc/openwin/lib/xil/config` takes precedence. You should edit the `/etc/openwin/lib/xil/config` file only.

This section explains how to create entries for compute device handlers, as well as how to add and delete these entries from the `config` file.

`config` *Entry*

This example shows a `config` text file entry for a dynamically loadable compute device handler.

```
#Start STOCKTICKERdevicename handler
class="XIL-COMPUTE" name="SOCKTICKERdevicename"
       priority="500" dependencies="STOCKTICKERiodevname";
#End STOCKTICKERdevicename handler
```

Formatting Guidelines

The `config` file has some basic formatting guidelines.

- Any characters following the pound character (#) through the end of a line are treated as a comment and are disregarded when the file is read.
- Quotation marks around value strings are required only if the string contains delimiters such as white space or a semicolon (;).
- The back slash character (\) can be used as an escape character. For example, \" is used to include the double quotation mark character as part of a string value.
- Parsing routines strip the quotation marks surrounding string values and pass the string only to the underlying software. The parsing software treats all values as strings; interpretation of the string value is up to the device handler.

By convention, the strings ‘Start’ and ‘End’ indicate the beginning and end of the database definitions for a set of loadable device handlers. In the example above (see “config Entry” on page 94) the string ‘STOCKTICKERdevicename handler’ identifies the STOCKTICKER device handler. The contents of the string you use are up to you. You use this string in a script file to append or delete your entry from the `config` file.

Between the ‘Start’ and ‘End’ strings are one or more loadable compute device handler entries for XIL. Each compute device handler description consists of up to four “attribute=value” pairs separated by white space (including tabs, spaces, and new line characters) and terminated by a semicolon.

Table describes each attribute=value pair.

Table 4-1 XIL Device Handler Attributes

Attribute=	**Meaning=**
class=	This value is always XIL-COMPUTE, which uniquely identifies the class of compute device handlers for XIL.
name=	This value is the name of the compute device. It uniquely identifies the instance of the class object in the `config` file and is different for each device handler. XIL derives the actual filename from this string as `xilcompute_`*name*`.so.2`.
priority=	This field is used by XIL to determine which compute device function to call when there are overlaps. Most compute devices will implement an atomic function already implemented by the default XIL memory pipelines. For example, a medical imaging pipeline may be accelerated single band “lookup 16->8” and no other lookup routines. Since it is a specialized routine taking advantage of hardware or a clever algorithm, you will want it called before the memory device routine. The priority for the memory pipelines are set at 100. We recommend device handler providers specify a priority of 500 unless they’re aware of other devices. A priority of “-1” will turn off the named pipeline such that it will not be loaded by XIL.
dependencies=	These are the names of I/O or compression devices which must be loaded *before* the described compute device can be loaded.

Using Script Files

Before your compute device handler can be loaded, you need to add an entry in `config`. To add (or delete) `config` entries, you should create an executable script file.

Appending An Entry

You add entry to `config` by appending the entry to the file. The following procedure illustrates this process.

1. **Create an entry.**
   Use the example below as a template to create your entry, filling in values for your device handler. Replace the string associated with ‘Start’ and ‘End’ with a string of your choice.

```
#Start STOCKTICKERdevicename
class=”XIL-COMPUTE” name=”STOCKTICKERdevicename”
       priority=”500” dependencies=”STOCKTICKERiodevname”;
#End STOCKTICKERdevicename
```

2. **Save your entry.**
   Save locally as `config`.

3. **Create a script file** `ins.config` **to append your entry.**
   Use the script file shown below as a template.

```
#! /bin/ksh
#
# Installation script for the config class
# If a config file existed, remove any entry belonging to
# this package, and append a new entry
#
echo $1
echo $2
chmod 644 $2
    if [ -r $2 ]
    then
        #It is editable by this script. Edit it.
        cp $2 /tmp/$$config || exit 2
       sed -e “/# Start STOCKTICKERdevicename/,/# End STOCKTICKERdevicename/d” \
        /tmp/$$config > $2 || exit 2
        cat $1 >> $2 || exit 2
        rm -f /tmp/$$config
    else
        #A config file was not present
        cat $1 >> $2 ||exit 2
        fi
chmod 444 $2
exit 0
```

4. **Execute the script file.**
   First you need to become superuser. The executable script file `ins.config` takes two arguments: the relative path to the local text file entry (`config`) and the full path to the system `config` file (`/etc/openwin/lib/xil/config`).

   As the script file executes, it displays the values (`config` and `/usr/openwin/lib/xil/config`) of the two arguments passed to it.

```
% su
Password:
# ./ins.config /etc/openwin/lib/xil/config
config
/etc/openwin/lib/xil/config
# ^D
```

Removing An Entry

To remove a `config` entry, use the following procedure.

1. **Use the script** `rm.config` **below as a template.**
   Replace the string ‘`STOCKTICKERdevicename`’ with the string for the entry you want to remove. Save the script with a name such as `rm.config`, and make the file executable.

```
#! /bin/ksh
#
# Removal script for the config class
# Remove entries that belong to this device handler
# Delete the file if it is empty
#
echo $1
chmod 644 $1
   sed -e “/# Start STOCKTICKERdevicename/,/# End STOCKTICKERdevicename/d” $1 > \
    /tmp/$$config || exit 2
    if [ -s /tmp/$$config ]
    then
    mv /tmp/$$config $1 || exit 2
    else
    rm $1 || exit 2
    fi
chmod 444 $1
exit 0
```

2. **Execute the script.**
   First become superuser. In this case, the script file `rm.config` takes one argument: the path to the system `config` file. As the script executes, it displays the value (`/etc/openwin/lib/xil/config`) of the single argument passed to it, as shown here.

```
% su
Password:
# ./rm.config /etc/openwin/lib/xil/config
/etc/openwin/lib/xil/config
#^D
```

4

Compute Device Handler- Basic Structure Variations

Compute device handlers follow the same basic structure as all operations in the XIL library. (See “Implementing an XIL Operation” on page 52 for a description of this basic structure.) Compute handlers also include special case operations such as convolution and geometric operations that involve variations on that basic structure.

The following sections take you through basic structure variations that are required for special case compute operations. Operations are presented in the following categories:

- Data collection (for example, `xil_histogram()`, `xil_extrema()`)
- Area-based (for example, `xil_convolve()`, `xil_erode()`)
- Geometric (for example, `xil_affine()`, `xil_scale()`)

Data Collection Operations

Data collection operations such as `xil_histogram()` take an image and generate data from that image to create a histogram. These operations operate in a multi-threaded fashion by having each compute routine produce data for the box list that they are handed and then reporting that data to the `XilOp` object which is responsible for collecting the data into a single entity and reporting the entity to the API user. The XIL core is responsible for coordinating the multiple calls to the compute routine and reporting the collected results back to the API user.

The compute routines report the data they have collected to the `XilOp` object using the `XilOp::reportResults()` function. `XilOp::reportResults()` takes a variable number of `void*` arguments, the types of which vary based on the compute routine. See Appendix A, “XilOp Object,” for details on the parameters passed back for `reportResults()`. The example below illustrates how the histogram compute routine uses this function.

```
    //
    //  Allocate and init histogram data array
    //
    unsigned int nElements = histogram->getNElements();
    unsigned int* data = new unsigned int[nElements];
```

```
    //
    // Code deleted for brevity

    XilStatus status = op->reportResults((void**)&data);
```

Area-Based Operations

This section covers area-based operations such as `xil_convolve()`, `xil_erode()`, `xil_dilate()`, `xil_fill()`, and `xil_error_diffusion()`. Unlike point-based operations such as `xil_add()`which generate a single destination pixel from a single source pixel, area-based operations generate a single destination pixel from multiple source pixels.

To do this, the compute routine needs to access source data outside of the pixel area defined by the operation.

XIL accomplishes this by modifying the source boxes passed in to the compute routine in the `XilBoxList` to ensure that enough storage exists to access all needed source pixels.

In a point-based operation, the source box represents the source pixels that correspond to the destination pixels being touched. In an area-based operation, the source box represents the area that corresponds to the destination pixels being touched—with an additional border region to provide the source pixels for processing the edge pixels of the destination area when there is enough data in the source image to do so.

The various area-based operations have slightly different requirements, each of which is detailed in this section.

Convolution, Erode, and Dilate

Convolution, erode, and dilate operations use an XilKernel or an XilSel object to describe how to combine source pixels in the destination pixel. For these operations, the call to `XilOp::splitOnTileBoundaries()` produces source boxes labeled as one of the following types:

- `XIL_AREA_TOP_LEFT_CORNER`
- `XIL_AREA_TOP_EDGE`
- `XIL_AREA_TOP_RIGHT_CORNER`
- `XIL_AREA_LEFT_EDGE`

- XIL_AREA_CENTER
- XIL_AREA_RIGHT_EDGE
- XIL_AREA_BOTTOM_LEFT_CORNER
- XIL_AREA_BOTTOM_EDGE
- XIL_AREA_BOTTOM_RIGHT_CORNER

Generation of these types is based on the position of the box within the source image, the dimensions of the XilKernel or XilSel object, and its key value.

As an example, Figure 4-1 shows a 5x5 kernel with a key value of 1,1. Edges represent the distance in each direction from the key value to the edge of the kernel. The edges determine how much extra storage is needed outside a given box and are referred to by their position: left, right, top, and bottom.

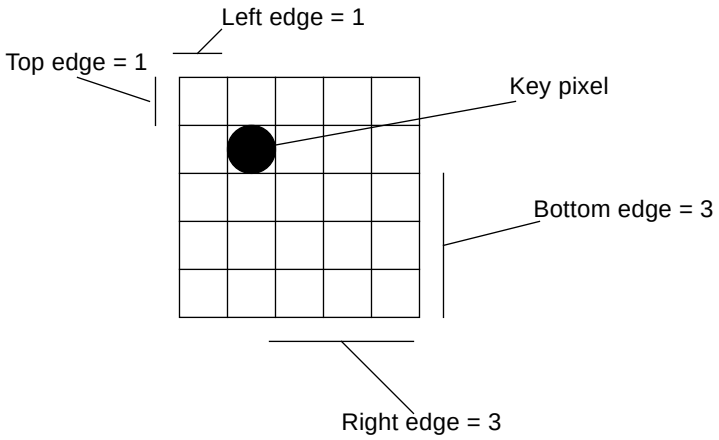


Figure 4-1 A 5x5 Kernel

Figure 4-2 illustrates a source image and shows all nine box types and their positions within the image. This layout is for the kernel shown in Figure 4-1.

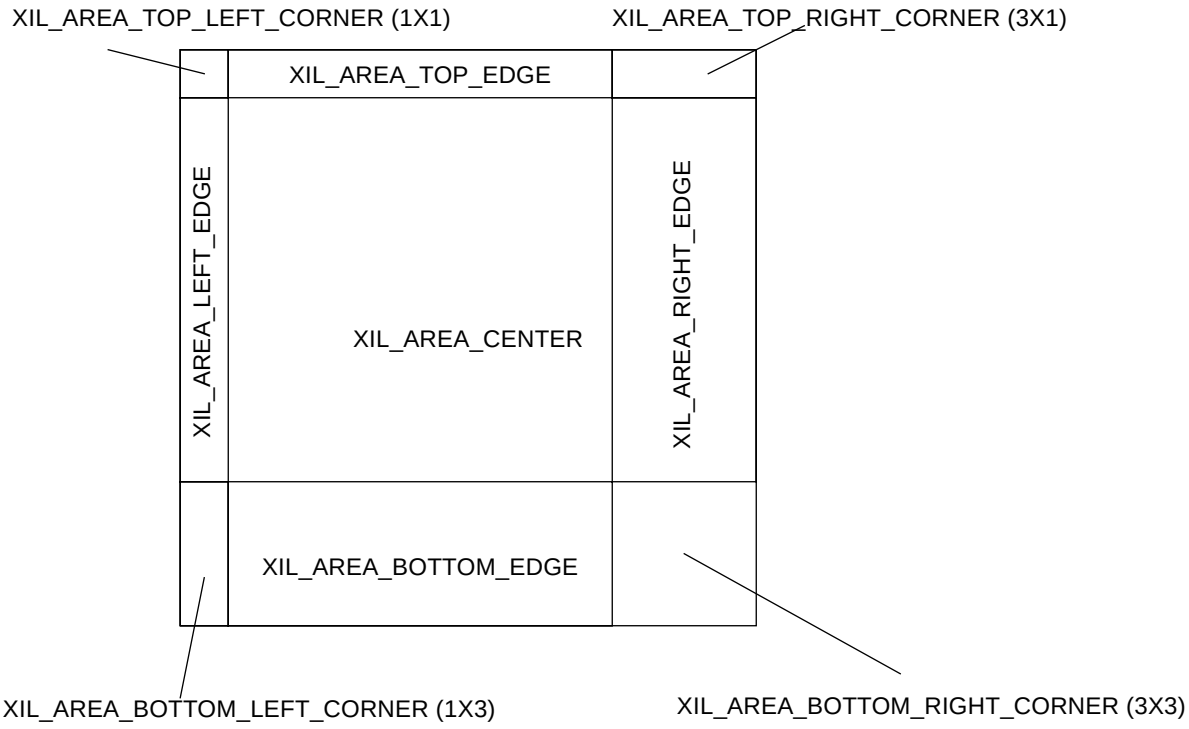


Figure 4-2 Box Types

Note – You cannot go (nor should ever need to go) outside the destination box.

Center Boxes

A *center* box (that is, XIL_AREA_CENTER) represents the simplest case for processing. A center box represents a source box which provides the required storage on all sides of the box. (See Figure 4-3.)

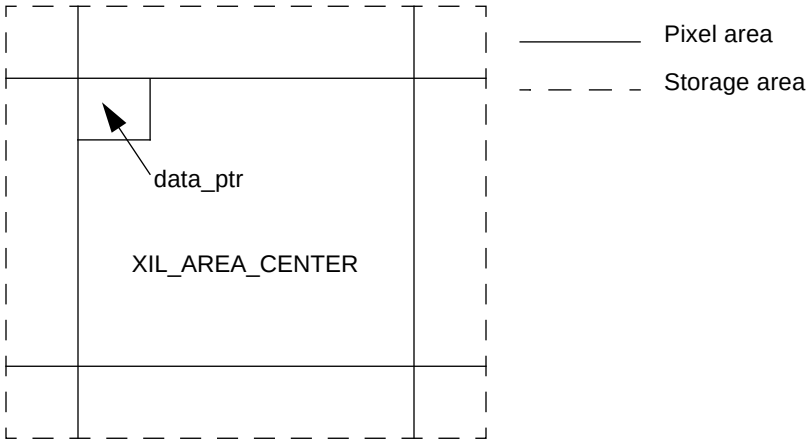


Figure 4-3 Center Box

Convolution and Edge Conditions

With convolution, the user may select one of the following edge condition options:

- XIL_EDGE_EXTEND
- XIL_EDGE_NO_WRITE
- XIL_EDGE_ZERO_FILL

See the *XIL Programmer's Guide* for more information.

When the API user selects the XIL_EDGE_NO_WRITE option, the XIL core only provides boxes of the XIL_AREA_CENTER type. The other two options produce edge and corner boxes for processing.

Note – As a protection against future versions of XIL that may add other edge conditions, the compute routine should test for and fail if it encounters an edge condition other than XIL_EDGE_EXTEND, XIL_EDGE_NO_WRITE, or XIL_EDGE_ZERO_FILL.

In Figure 4-3, the solid line box shows the pixel area in the source, and the dashed line shows the storage available outside the box. When requesting the storage information for the box, the XIL core sets the data pointer at the top left hand pixel coordinate. The compute routine can then be guaranteed that it can place the kernel key in the top left hand corner of the box to start computing the destination pixel, knowing also that when it reaches the bottom edge of the box, source storage is available to correctly compute the destination area. No edge handling is needed for this case.

Edge Boxes

An *edge* box (`XIL_AREA_TOP`, `XIL_AREA_BOTTOM`, `XIL_AREA_LEFT`, `XIL_AREA_RIGHT`) differs from a center box in that it only has source storage available on three sides. Figure 4-4 represents an `XIL_AREA_LEFT_EDGE` box

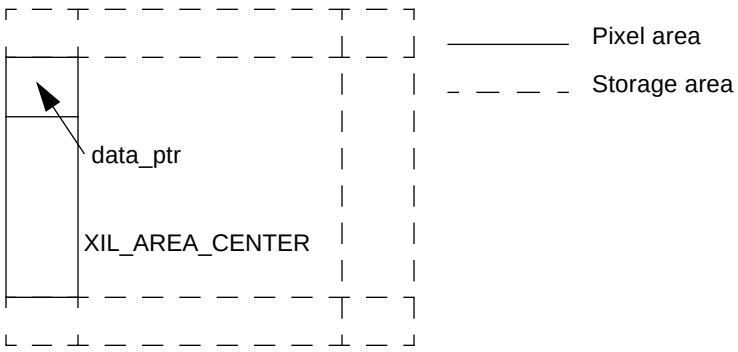


Figure 4-4 `XIL_AREA_LEFT_EDGE` Box

Again, the solid line represents the pixel area and the dashed line, the provided storage. The box follows the edge of the image in one dimension (unless limited by a ROI) and is the width of the appropriate kernel edge in the other. In the case of the `XIL_AREA_LEFT_EDGE` box, the height of the box (assuming no ROI) is the height of the image minus the `TOP_LEFT_CORNER` and `BOTTOM_LEFT_CORNER` boxes. The width of the box is 1.

The data pointer again is returned at the top left pixel coordinate but, in this case, no storage is available for the left hand side of the kernel.

The compute routine must process the pixels represented by this box according to the rules of the operation or, in the case of convolution, according to the edge condition selected by the API user.

Corner Boxes

A *corner* box (`XIL_AREA_TOP_LEFT_CORNER`, `XIL_AREA_TOP_RIGHT_CORNER`, `XIL_AREA_LOWER_LEFT_CORNER`, `XIL_AREA_LOWER_RIGHT_CORNER`) is a degenerate case of an edge box. Two sides of a corner box cannot provide enough source data for the pixel area to be processed. Figure 4-5 illustrates the case of `XIL_AREA_TOP_LEFT_CORNER`.

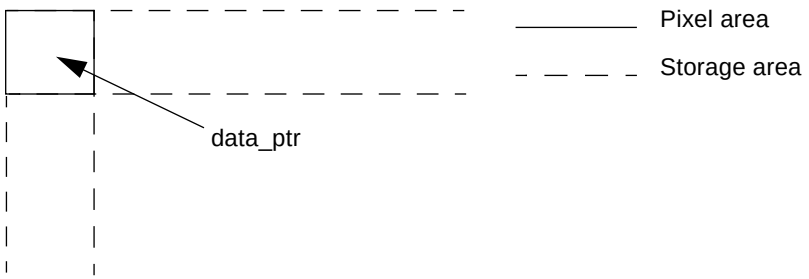


Figure 4-5 Corner Boxes

A corner box is guaranteed to be the size of the kernel edges for the given corner. In the case of Figure 4-6 with the kernel defined in Figure 4-1, the size of the `XIL_AREA_TOP_LEFT_CORNER` is 1x1.

As with edge boxes, the compute routine is responsible for processing the box according to the rules of the operation.

Getting The Box Type

To get the type of the box, use the `getBoxTag()` function. This function returns a `void*` which, when cast to an `XilAreaBoxType`, provides the box type. The same tag is applied to both the source and destination box.

```
XilBoxAreaType tag = (XilBoxAreaType) src_box->getTag();
switch (tag) {
  case XIL_AREA_TOP_LEFT_CORNER:
  case XIL_AREA_TOP_EDGE:
  case XIL_AREA_TOP_RIGHT_CORNER:
  case XIL_AREA_RIGHT_EDGE:
  case XIL_AREA_CENTER:
  case XIL_AREA_LEFT_EDGE:
  case XIL_AREA_BOTTOM_LEFT_CORNER:
```

```
XilBoxAreaType tag = (XilBoxAreaType) src_box->getTag();
  case XIL_AREA_BOTTOM_EDGE:
  case XIL_AREA_BOTTOM_RIGHT_CORNER:
    break;
```

Although destination boxes have tags, the tag only has meaning with regard to the source box.

Performance Considerations

It is optional to call `XilOp::splitOnTileBoundaries()` at the beginning of a compute routine. However, this has additional ramifications in area operations.

In the general case, `splitOnTileBoundaries()` ensures that all source boxes in the `XilBoxList` do not cross tile boundaries. To do so, it may have to split the passed-in boxes into smaller regions. This means, when `getStorage()` is subsequently called for each set of boxes in the box list, the core never has to cobble tiled regions of data for access by the compute routine.

In the area-operation case, `splitOnTileBoundaries()`not only splits the boxes along source tiles, it secondarily splits those boxes as needed to indicate edge or corner conditions.

To process pixels in the destination corresponding to source pixels that lie along source tile boundaries, the compute routine must access pixels from the neighboring tile. To minimize cobbling, `splitOnTileBoundaries()`generates boxes just large enough to handle these tile edges. As long as these boxes do not lie on source image edges, the boxes are tagged as `XIL_AREA_CENTER` and are processed by the compute routine as any other center box.

If the compute routine chooses not to call `splitOnTileBoundaries()`, the box passed in represents the entire destination region to be processed and the corresponding source region, regardless of source tiles. None of the boxes are tagged, and it is up to the compute routine to identify those destination pixels for which there may not be enough source data. In such cases it should not try to access outside of the source image.

Kernel Inversion

In the case of convolution, the kernel that the user generates is inverted before being passed to the compute routine. This is a difference from previous XIL releases in which the compute routine was responsible for inverting the kernel.

Note – This is convenient for the compute routine since the kernel is stored on the op in the form needed for convolution processing.

Fill and Error Diffusion

Fill and error diffusion operations are handled in a similar manner. Currently these operations are called from a single thread. The destination box supplied is the same as for the basic case (that is, it corresponds to the destination area to be written). The source box pixel area corresponds to the basic case with the exception that the storage is available for the entire source image. This makes it possible for the compute routine to always have access to the entire source image data. The data pointer returned from

`XilStorage::getStorageInfo()` points to the box pixel location. The box location within the image can be retrieved using the `XilBox::getAsRect()` function. Figure 4-6 illustrates the source box setup for fill and error diffusion.

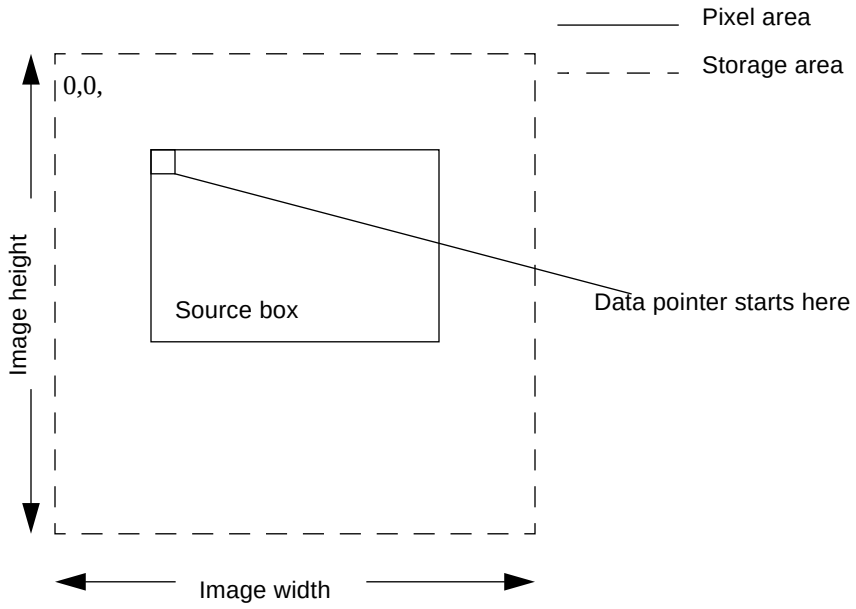


Figure 4-6 Fill and Error Diffusion Source Box Setup

Geometric Operations

Geometric operations such as affine, transpose, rotate, scale, translate, and tablewarp manipulate the image in some geometric fashion. This means that the mapping of a destination pixel to a pixel in the source is dependent on the parameters of the operation. Implementation of each of these functions varies from the implementation of the basic case described in “Basic Structure: Atomic Function” on page 54.

This section first discusses the transpose operation as it introduces the concept common to all geometric compute routines of backward mapping from a destination area to get the corresponding area in the source. The affine operation builds on this concept and introduces two new objects, the `XilConvexRegionList` and the `XilScanLineList`.

The tablewarp operation is the most complex. The compute routine is required to handle almost all the computation, because the XIL core has no knowledge of how a destination pixel maps back to the source.

Transpose

The transpose operation is a *point operator* in the sense that it requires only one source pixel to generate each destination pixel. However, which source pixel corresponds to which destination pixel is dependent on the flip-type provided to the operation.

The source box passed in to the transpose routine maps exactly to the destination box. The compute routine then needs to calculate the mapping of the pixels within one box to the pixels in the other. The compute routine may choose to map destination to source pixels for each flip-type explicitly or to use the `XilOp::backwardMap()` routine provided by XIL for convenience, as shown here.

The full interface can be found in `_XilOp.hh`

```
//
//  Backward map a single point in destination box space to the
//  corresponding point in source box space.  The last (optional)
//  argument indicates which source to backward map into.
//
XilStatus backwardMap(XilBox*       dst_box,
                      float         dx,
                      float         dy,
                      XilBox*       src_box,
                      float*        sx,
                      float*        sy,
                      unsigned int  src_number = 1);
```

As in the base case (see the subsection entitled“Step 5: Processing the Data” on page 59” in “Basic Structure: Atomic Function”), the transpose routine generates an `XilRectList` from the passed-in ROI and destination box. It is then responsible for correctly copying the equivalent source rectangles into the destination based on the flip type. As before, the data pointers returned in the storage object from `XilImage::getStorage()` are relative to the upper left corner of each of the boxes.

The following example shows how the `backwardMap()` function could be used in the memory transpose compute routine to get the correct source area.

```
//
//  Create a list of rectangles to loop over.  The resulting list
//  of rectangles is the area created by intersecting the ROI with
//   the destination box.
//
XilRectList    rl(roi, dst_box);
//
// loop over the list of rectangles
//
while (rl.getNext(&dstR_x, &dstR_y, &dstR_xsize, &dstR_ysize)) {
    //
    // The rectangle in the list applies to the dst, so we have
    // to find appropriate rectangle in the src according to
    // the fliptype.
    //
    // The op does the backward map for us.
    //
        {
            float srcx;
            float srcy;
            op->backwardMap(dst_box, dstR_x, dstR_y,
                            src_box, &srcx, &srcy);
            srcR_x = (int)srcx;
            srcR_y = (int)srcy;
        }
    src_scanline = src_data + (srcR_y * src_scanline_stride)
         + (srcR_x * src_pixel_stride);
    dst_scanline = dst_data + (dstR_y * dst_scanline_stride)
         + (dstR_x * dst_pixel_stride);

    //
    // Note that the compute routine will still have to take into
    // account incrementing the source in the appropriate
    // direction for the flip-type.
    //
```

Affine

In an affine operation, backward mapping a rectangle from the destination to the source may generate a region that is not a rectangle. In order to support this, the XIL GPI represents an ROI as an `XilConvexRegionList`.

A *convex region* is defined as a convex polygon whose points are stored as floating-point values. The convex region is represented as two arrays of floating-point values: one for the x-points and the other, forthe y-points. A `point_count` indicates the length of the arrays, as shown here. The full interface can be found in `_XilConvexRegionList.hh`.

```
//
//  Construction using a roi and a box. The clipped regions in
//  the list are all relative to (0, 0) in the box
//
XilConvexRegionList(XilRoi* roi, XilBox* dest_box);
//
//  Get the next convex region on the list
//
Xil_boolean getNext(const float** x_array,const float** y_array,
    unsigned int* point_count);

//
//  Allows the original full ROI convex region list to be
//  clipped by a different box
//
XilStatus reinit(XilBox* dest_box);
```

As with the `XilRectList`, we recommend creating the `XilConvexRegionList` on the stack to minimize the use of `new()` and `delete()` within the compute routine. The compute routine then loops through all convex regions in the list processing each one, until `XilConvexRegionList::getNext()` returns `NULL`.

To process a given region, the compute routine must backward map the destination region to the equivalent source region. This can be done by the compute routine with the affine matrix on the op, or it can be done with `XilOp::backwardMap()`.

As with area operators, the source boxes provided to the compute routine have enough storage associated with them to provide all the data for the given interpolation type. The destination box is small enough to guarantee enough source storage is available for interpolating along all destination edges.

Once the compute routine has obtained a destination region and mapped it to the corresponding source region, it is responsible for moving the data pointers to the correct positions within the boxes and for generating the destination pixels using the appropriate interpolation kernel.

For convenience, XIL provides the `XilScanLineList` object, which can be used to turn a convex region into a list of scanlines, as shown here. The full interface can be found in `_XilScanlineList.hh`.

```
XilScanlineList(const float* x_array,const float*
                y_array,unsigned int num_points);

//
//  Get the next scanline.
//
Xil_boolean    getNext(int*            y,
                       float*          x1,
                       float*          x2);

Xil_boolean    getNext(unsigned int*   y,
                       unsigned int*   x1,
                       unsigned int*   x2);

//
// Return the number of scanlines in the list
//
unsigned int   getNumScanlines();
```

The following example show the use of of the `XilConvexRegionList` and the `XilScanLineList`.

```
//
//  Loop over convex regions in the destination.
//
XilConvexRegionList crl(roi, dst_box);

unsigned int  num_pts;
const float* dst_xarray;
const float* dst_yarray;
while(crl.getNext(&dst_xarray, &dst_yarray, &num_pts)) {
    //
    //  Create scanline list
    //
    XilScanlineList dst_scanlines(dst_xarray, dst_yarray,
                                  num_pts);

    //
    //  Loop over scanlines.
    //
    unsigned int y;
    unsigned int dst_scan_start;
    unsigned int dst_scan_end;
    while(dst_scanlines.getNext(&y, &dst_scan_start,
          &dst_scan_end)) {
        //
        // Backward map the two endpoints of the line
        //
       op->backwardMap(dst_box, (float)dst_scan_start, (float)y,
                       src_box, &start_x, &start_y);
        op->backwardMap(dst_box, (float)dst_scan_end, (float)y,
                       src_box, &end_x, &end_y);

        //
        // process destination line, walking the source line
        //
```

Kernel Definitions

The edge areas available outside the source pixel retion are defined as follows:

1. Nearest neighbor: Left = Right = Top = Bottom = 0

2. Bilinear: Left = Top =0
3. Bicubic: Left = Top = 1
4. General: Values are calculated based on the size of the horizontal interpolation table width and size of the vertical interpolation table height using these equations.

```
unsigned int keyX = (width - 1)/2;
unsigned int keyY = (height - 1)/2;
   Left = keyX;
   Right = width - (keyX + 1);
   Top = keyY;
   Bottom = height - (keyY + 1);
```

Performance Considerations

The destination box does not cross-tile boudnaries. The corresponding source box may cross tile boundaries, and the compute routine may choose to call `XilOp::splitOnTileBoundaries()` to split the boxes in the original box list along source tile boundaries. Because of the area-based nature of affine operations, some source boxes must cross source tile boundaries. These boxes are made as small as possible to minimize source data cobbling.

Note – `XilOpAffine::splitOnTileBoundaries()` currently does nothing.

Rotate

Rotate is very similar to affine. In fact, the XIL core treats the two operations in the same manner. (See “Affine” on page 112 for details.)

Scale and Translate

Scale and translate operations are special cases of affine that allow you to use rectangles rather than convex regions. Although the compute routine can use `XilConvexRegionList`, performance can be enhanced if it instead uses `XilRectList` to generate the destination regions. The compute routine can use `XilOp::backwardMap()` to generate the equivalent source rectangle.

Tablewarp

In the case of the the `xil_tablewarp()`, `xil_tablewarp_vertical()`, and `xil_tablewarp_horizontal()` functions, the compute routine is responsible for handling all the backward mappings from the destination box through the tablewarp image to the source image.

For tablewarp, the ROI passed in to the compute routine is just the ROI of the destination image. The source ROI is passed in as a separate argument. It is the responsibility of the compute routine to position the warp image correctly on the destination ROI and to take the source ROI into account when calculating the backward mapping position. The source box passed in to the compute routine represents the whole source image, since mapping of destination to source pixels is known.

Compression/Decompression 5

This chapter is TBD.

5

Storage Devices 6

About Storage Devices

This chapter is TBD.

6

`XilOp` *Object*

A

This appendix identifies the source images, destination image, and parameters supported by each XIL atomic function. In addition it explains how to extract the images and parameters. You must know this information anytime you implement XIL atomic functions, such as when you write a compute device handler.

When writing a molecule, you are responsible for retrieving the appropriate `XilOp` object from the op list in order to access each `XilOp` object's parameters. For more information on atomic functions and molecules, see the respective sections:

- "Operation Prototype: Atomic Function" on page 53
- "Operation Prototype: Molecule" on page 62

Extracting Images and Parameters

To extract the images and parameters for an XIL atomic function, use the `XilOp` member functions described below. For more information on the `XilOp` class, see Chapter 1, "Overview."

Extracting Source Images

To get source (src) images use the method,

```
XilOp::getSrcImage(unsigned int image_number)
```

For example, to get the first source image,

```
op->getSrcImage(1)
```

Note – `image_number` starts at 1, not 0.

Extracting Destination Images

To get the destination (dst) image use the method,

`XilOp::getDstImage(unsigned int image_number)`

Currently, `image_number` is always 1.

Extracting Parameters

To get the function parameters, use the method that supports that parameter type. Each method is shown here.

```
XilOp::getParam(unsigned int param_number, int* param);
XilOp::getParam(unsigned int param_number, long long* param);
XilOp::getParam(unsigned int param_number, float* param);
XilOp::getParam(unsigned int param_number, double* param);
XilOp::getParam(unsigned int param_number, void** param);
XilOp::getParam(unsigned int param_number, XilObject** param);
```

To use the parameter methods, you are required to obtain the parameters in the same format in which they were stored. You must always retrieve pointers as `void**`. For example, to get a lookup table for `xil_lookup()` functionality,

```
XilLookupTable* lut
op->getParam(1, (XilObject**) &lut);
```

Source Images, Destination Image, and Parameters

The table below alphabetically presents the XIL atomic functions and lists the applicable source images, destination image, parameters supported by each, as well as the parameter type for retrieval.

XIL Function	Src 1	Src 2	Src 3	Dst	Parameters	getParam() Type
absolute	●			●		
add	●	●		●		
add_const	●			●	const_array[nbands] 1 Xil_signed8 -1 to 1 8 Xil_signed16 -255 to 255 16 Xil_signed32 -65535 to 65535 f32 float values passed on	void*
affine	●			●	float matrix[6[float xoffset float yoffset / / General Interpolation Only XilInterpolationTable horiz_tbl XilInterpolationTable vert_tbl	void* float float XilObject* XilObject*
and	●	●		●		
and_const	●			●	const_array[nbands] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 0 to 255 16 Xil_unsigned16 0 to 32767	void*
black_generation	●			●	float black float undercolor	float float
band_combine	●			●	XilKernel kernel	XilObject*
capture	●			●	In place operation src == dst src is the controlling image unsigned int bands_written; unsigned int offset_band;	 unsigned int unsigned int
blend	●	●	●	●	Src3 is the alpha image	

XIL Function	Src 1	Src 2	Src 3	Dst	Parameters	getParam() Type
color_convert	●			●	XilColorspace src_colorspace XilColorspace dst_colorspace	XilObject* XilObject*
color_correct	●			●	XilColorspaceList list	XilObject*
compress	●			●	Dst is an XilCis int write_frame	 int
cast	●			●		
convolve	●			●	XilKernel kernel XilEdgeCondition edge_condition	XilObject* void*
copy	●			●		
copy_pattern	●			●		
copy_with_planemask	●			●	const_array[nbands] 1 unsigned int 0 to 1 8 unsigned int 0 to 255 16 unsigned int 0 to 32767	void*
decompress	●			●	src1 is an XilCis int read_frame	 int
dilate	●			●	XilSel sel	XilObject*
divide	●	●		●		
display	●			●	In place operation src == dst src is the controlling image unsigned int bands_written; unsigned int offset_band;	 unsigned int unsigned int
divide_by_const	●			●	float const_array[nbamds]	void*
divide_into_const	●			●	const_array[nbands] 1 unsigned int 0 to 1 8 unsigned int 0 to 255 16 unsigned int 0 to 32767 f32 value passed on	void*
edge_detection	●			●	XilEdgeDetectionType type	void*
erode	●			●	XilSel sel	XilObject*

XIL Function	Src 1	Src 2	Src 3	Dst	Parameters	getParam() Type
error_diffusion	●			●	XilLookup colormap XilKernel distribution	XilObject* XilObject*
extrema	●				/ / reportResults parameters float max[nbands] float min[nbands]	 void* void*
fill	●			●	unsigned int xseed unsigned int yseed boundary[nbands] fill[nbands] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 (rounded) 16 Xil_unsigned16 (rounded) f32 Xil_float32	unsigned int unsigned int void* void*
histogram	●				XilHistogram histogram unsigned int skip_x unsigned int skip_y / / reportResults parameters unsigned int data[]	XilObject* unsigned int unsigned int void*
lookup	●			●	XilLookup lut	XilObject*
max	●			●		
min	●			●		
multiply	●	●		●		
multiply_const	●			●	float const_array[nbands]	void*
nearest_color	●			●	XilLookup cmap	XilObject*
not	●			●		
or	●	●		●		
or_const	●			●	const_array[nbands] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 0 to 255 16 Xil_unsigned16 0 to 32767	void*

XIL Function	Src 1	Src 2	Src 3	Dst	Parameters	getParam() Type
ordered_dither	●			●	XilLookup colormap XilDitherMask dithermask	XilObject* XilObject*
paint	●			●	float color[nbands] XilKernel brush unsigned int count unsigned int points[count]	void* XilObject* unsigned int void*
rescale	●			●	float scale_array[nbands] float offset_array[nbands]	void* void*
rotate	●			●	float angle float src_xoffset float src_yoffset / / General Interpolation only XilInterpolationTable horiz XilInterpolationTable vertical	float float float XilObject* XilObject*
scale	●			●	float xfactor float yfactor / / General Interpolation only XilInterpolationTable horiz XilInterpolationTable vertical	float float XilObject* XilObject*
set_value				●	const_array[nands] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 16 Xil_signed16 f32 Xil_float32	void*

XIL Function	Src 1	Src 2	Src 3	Dst	Parameters	getParam() Type
soft_fill	●			●	unsigned int xseed unsigned int yseed	unsigned int unsigned int
					foreground[nbands] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 16 Xil_signed16 f32 Xil_float32	void*
					unsigned int num_bgcolor	unsigned int
					background[num_bgcolor] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 16 Xil_signed16 f32 Xil_float32	void*
					fill_color[nbands] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 16 Xil_signed16 f32 Xil_float32	void*
squeeze_range	●				/ / reportResults int imax int imin Xil_unsigned8 result_flags[]	 int int void*
subsample_binary_to_gray	●			●	float xfactor float yfactor	float float
subsample_adaptive	●			●	float xfactor float yfactor	float float
subtract	●	●		●		
subtract_from_const	●			●	const_array[nbands] 1 Xil_signed8 0 to 2 8 Xil_signed16 0 to 510 16 Xil_signed32 -65536 to 65534 f32 values passed on	void*

XIL Function	Src 1	Src 2	Src 3	Dst	Parameters	getParam() Type
tablewarp tablewarp_horizontal tablewarp_vertical	●	●		●	Src2 is the warp_table image unsigned int src_xoffset unsigned int src_yoffset unsigned int dst_xoffset unsigned int dst_yoffset unsigned int warp_xoffset unsigned int warp_yoffset XilRoi* src_image_roi / / General interpolation only XilInterpolationTable horiz XilInterpolationTable vert table	 unsigned int unsigned int unsigned int unsigned int unsigned int unsigned int XilObject* XilObject* XilObject*
threshold	●			●	low[nbands] high[nbands] map[nbands] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 (values rounded) 16 Xil_signed16 (values rounded) 32 Xil_float32 (values passed on)	void* void* void*
translate	●			●	float xoffset float yoffset / / General interpolation only XilInterpolationTable horiz XilInterpolationTable vert	float float XilObject* XilObject*
transpose	●			●	XilFlipType fliptype	void*
xor	●	●		●		
xor_const	●			●	const_array[nbands] 1 Xil_unsigned8 0 to 1 8 Xil_unsigned8 0 to 255 16 Xil_unsigned16 0 to 32767	void*

XIL Atomic Functions B

Table B-1 lists the XIL atomic functions. The syntax for these entries is,

```
<operation name>;<src1 data type>,<src2 data type>,<src3 data type>-><dst data type>
```

If the operation uses of all the same data type this long form is shortened to, `<operation name>;<data type>`.

where *data type* is one of the following,

1. 1 - 1-bit images
2. 8 - 8-bit images
3. 16 - 16-bit images
4. f32 - 32-bit floating point images

For example for the add operation for 16 bit data the atomic function is,

```
add;16
```

The API call `xil_blend()` for 32-bit floating point data with a 16-bit blend image, illustrates the full syntax.

```
blend;f32,f32,16->f32
```

B

The first column gives the name of the function that must be supplied in the `config` file header comment in order to associate an implemented function with an API call. The second column lists the data types for which the operation is valid. The third column gives the name of the API binding call associated with the atomic name. Further description of these API functions can be found in the *XIL Reference Manual*.

Table B-1 XIL Atomic Functions *(1 of 5)*

Atomic Function	Valid Data Combinations	API Call
absolute	16 f32	xil_absolute
add	1 8 16 f32	xil_add
add_const	1 8 16 f32	xil_add_const
affine_bicubic	1 8 16 f32	xil_affine, bicubic interpolation
affine_bilinear	1 8 16 f32	xil_affine, bilinear interpolation
affine_general	1 8 16 f32	xil_affine, general interpolation
affine_nearest	1 8 16 f32	xil_affine, nearest neighbor interpolation
and	1 8 16	xil_and
and_const	1 8 16	xil_and_const
band_combine	1 8 16 f32	xil_band_combine
black_generation	1 8 16 f32	xil_black_generation

Table B-1 XIL Atomic Functions *(2 of 5)*

Atomic Function	Valid Data Combinations	API Call
blend	1,1,1->1 1,1,8->1 1,1,16->1 1,1,f32->1 8,8,1->8 8,8,8->8 8,8,16->8 8,8,f32->8 16,16,1->16 16,16,8->16 16,16,16->16 16,16,f32->16 f32,f32,1>f32 f32,f32,8->f32 f32,f32,16>f32 f32,f32,f32->f32	xil_blend
choose_colormap	8	xil_choose_colormap
color_convert	1 8 16 f32	xil_color_convert
color_correct	8	xil_color_correct
cast	1->8 1->16 1->f32 8->1 8->16 8->f32 16->1 16->8 16->f32 f32->1 f32->8 f32->16	xil_cast
convolve	1 8 16 f32	xil_convolve
copy	1 8 16 f32	xil_copy
copy_with_planemask	1 8 16	xil_copy_with_planemask
copy_pattern	1 8 16 f32	xil_copy_pattern
dilate	1 8 16 f32	xil_dilate
divide	1 8 16 f32	xil_divide
divide_into_const	1 8 16 f32	xil_divide_into_const
edge_detection	1 8 16 f32	xil_edge_detection
erode	1 8 16 f32	xil_erode

B

Table B-1 XIL Atomic Functions *(3 of 5)*

Atomic Function	Valid Data Combinations	API Call
error_diffusion	1->8 1->16 8->1 8->16 16->1 16->8 f32->1 f32->8 f32->16	xil_error_diffusion
extrema	1 8 16 f32	xil_extrema
fill	1 8 16 f32	xil_fill
histogram	1 8 16 f32	xil_histogram
lookup	1->1 1->8 1->16 1->f32 8->1 8->8 8->16 8->f32 16->1 16->8 16->16 16->f32	xil_lookup
max	1 8 16 f32	xil_max
multiply	1 8 16 f32	xil_multiply
multiply_const	1 8 16 f32	xil_multiply_const
nearest_color	1->1 1->8 1->16 8->1 8->8 8->16 16->1 16->8 16->16 f32->1 f32->8 f32->16	xil_nearest_color
not	1 8 16	xil_not
or	1 8 16	xil_or
or_const	1 8 16	xil_or_const
ordered_dither	1->1 1->8 1->16 8->1 8->8 8->16 16->1 16->8 16->16 f32->1 f32->8 f32->16	xil_ordered_dither
paint	1 8 16 f32	xil_paint
rescale	1 8 16 f32	xil_rescale
rotate_bicubic	1 8 16 f32	xil_rotate, bicubic interpolation
rotate_bilinear	1 8 16 f32	xil_rotate, bilinear interpolation
rotate_general	1 8 16 f32	xil_rotate, general interpolation

Table B-1 XIL Atomic Functions *(4 of 5)*

Atomic Function	Valid Data Combinations	API Call
rotate_nearest	1 8 16 f32	xil_rotate, nearest neighbor interpolation
scale_bicubic	1 8 16 f32	xil_scale, bicubic interpolation
scale_bilinear	1 8 16 f32	xil_scale, bilinear interpolation
scale_general	1 8 16 f32	xil_scale, general interpolation
scale_nearest	1 8 16 f32	xil_scale, nearest neighbor interpolation
set_value	1 8 16 f32	xil_set_value
separable_convolve	1 8 16 f32	xil_convolve with a separable kernel
soft_fill	1 8 16 f32	xil_soft_fill
squeeze_range	1 8 16	xil_squeeze_range
subsample_binary_to_gray	1->8	xil_subsample_binary_to_gray
subsample_adaptive	1 8 16 f32	xil_subsample_adaptive
subtract	1 8 16 f32	xil_subtract
subtract_from_const	1 8 16 f32	xil_subtract_from_const
tablewarp_bicubic	1 8 16 f32	xil_tablewarp, bicubic interpolation
tablewarp_bilinear	1 8 16 f32	xil_tablewarp, bilinear interpolation
tablewarp_general	1 8 16 f32	xil_tablewarp, general interpolation
tablewarp_nearest	1 8 16 f32	xil_tablewarp, nearest neighbor interpolation
tablewarp_horizontal_bicubic	1 8 16 f32	xil_tablewarp_horizontal, bicubic interpolation
tablewarp_horizontal_bilinear	1 8 16 f32	xil_tablewarp_horizontal, bilinear interpolation

B

Table B-1 XIL Atomic Functions *(5 of 5)*

Atomic Function	Valid Data Combinations	API Call
tablewarp_horizontal_general	1 8 16 f32	xil_tablewarp_horizontal, general interpolation
tablewarp_horizontal_nearest	1 8 16 f32	xil_tablewarp_horizontal, nearest neighbor interpolation
tablewarp_vertical_bicubic	1 8 16 f32	xil_tablewarp_vertical, bicubic interpolation
tablewarp_vertical_bilinear	1 8 16 f32	xil_tablewarp_vertical, bilinear interpolation
tablewarp_vertical_general	1 8 16 f32	xil_tablewarp_vertical, general interpolation
tablewarp_vertical_nearest	1 8 16 f32	xil_tablewarp_vertical, nearest neighbor interpolation
threshold	1 8 16 f32	xil_threshold
translate_bicubic	1 8 16 f32	xil_translate, bicubic interpolation
translate_bilinear	1 8 16 f32	xil_translate, bilinear interpolation
translate_general	1 8 16 f32	xil_translate, general interpolation
translate_nearest	1 8 16 f32	xil_translate, nearest neighbor interpolation
transpose	1 8 16 f32	xil_transpose
xor	1 8 16	xil_xor
xor_const	1 8 16	xil_xor_const

Index